COMPOSING HYBRID DISCRETE EVENT SYSTEM AND CELLULAR AUTOMATA MODELS

by

Gary R. Mayer

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

August 2009

COMPOSING HYBRID DISCRETE EVENT SYSTEM

AND CELLULAR AUTOMATA MODELS

by

Gary R. Mayer

has been approved

April 2009

Graduate Supervisory Committee:

Hessam S. Sarjoughian, Chair
C. Michael Barton
Marcus A. Janssen
Gregory M. Nielson

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Hybrid system models — those devised from two or more disparate sub-system models — provide a number of benefits in terms of conceptualization, development, and assessment of dynamical systems. The decomposition approach helps to formulate complex interactions that are otherwise difficult or impractical to express. However, hybrid model development and usage can introduce complexity that emerges from the composition itself. To improve assurance of model correctness, sub-systems using disparate modeling formalisms must be integrated above and beyond just the data and control level; their composition must have model specification and simulation execution aspects as well. Poly-formalism composition is one approach to composing models in this manner.

This dissertation describes a poly-formalism composition between a Discrete EVent System specification (DEVS) model and a Cellular Automata (CA) model types. These model specifications have been chosen for their broad applicability in important and emerging domains. An agent-environment domain exemplifies the composition approach. The inherent spatial relations within a CA make it well-suited for environmental representations. Similarly, the component-based nature of agents fits well within the hierarchical component structure of DEVS. This composition employs the use of a third model, called an interaction model, that includes methods for integrating the two model types at a formalism level, at a systems architecture level, and at a model execution level. A prototype framework using DEVS for the agent model and GRASS for the environment has been developed and is described. Furthermore, this dissertation explains how the concepts of this composition approach are being applied to a real-world research project.

This dissertation expands the tool set modelers in computer science and other disciplines have in order to build hybrid system models, and provides an interaction model for an

on-going research project. The concepts and models presented in this dissertation demonstrate the feasibility of composition between discrete-event agents and discrete-time cellular automata. Furthermore, it provides concepts and models that may be applied directly, or used by a modeler to devise compositions for other research efforts.

This thesis is dedicated to my wife, Kristin.

This is as much a result of my work as your love and support. Thank you.

This thesis is also dedicated to the memory of my father, Mark Mayer, to the memory of my grandmother, Gertrude Mayer, and to the memory of my my great-uncle, Milton Simbrow. The three of them are very much the reason for who I am today.

ACKNOWLEDGMENTS

Foremost, I would like to acknowledge the selflessness, help, support, guidance, patience, and friendship of my advisor, Dr. Hessam Sarjoughian. A man unique amongst his peers — and that's a good thing. He has taught me modeling and simulation, and provided me goals to strive for in my own career as a professor. I greatly appreciate him having the flexibility and patience to allow me the opportunity to disagree. In that regard, I must also extend my thanks and apologies to his wife, Nazilla. The many hours "after hours" we discussed the finer points of modeling and simulation often got Hessam home too late. But, it also helped the two of us grow in our understanding of the subject.

I would also like to thank my committee members: Dr. C. Michael Barton, Dr. Marco Janssen, and Dr. Gregory Nielson. Their feedback and support is greatly appreciated. It has helped with both the completion of my research, the completion of this dissertation, and moving onto my next career.

Next, I must extend my gratitude to another friend and former advisor, Dr. Jerry Weinberg at Southern Illinois University Edwardsville. Jerry set me on a path of academia and encouraged me to pursue my PhD. He is another professor whose approach to students and education I hope to model in my own career.

My sincere appreciation for their patience and guidance goes out to Dr. Barton, Mr. Isaac Ullah, Ms. Eowyn Allen, and Mr. Sean Bergin as I sought to learn about their world and they, mine. This is especially true of Eowyn. Eowyn, I cherish the friendship we formed and appreciate being able to share the ups and the downs. May you find happiness in this world and forever keep hold of it.

Next, I must express my gratitude to all of the past and present members of the ASU-ACIMS lab with whom I have had the pleasure of interacting on a regular basis. Dr.

Weilong Hu, Dr. Dongping Huang, Mr. Robert Flasher, Mr. Charles Dairman, Mr. Sung-Ung Kim, Mr. Vignesh Elamvazhuthi, Mr. Sajjan Sarkar, Mr. Mohammed Muqsith, Mrs. Kahkashan Shaukat (yes, your as much a part of the lab as the rest), Mr. Jonathan Gibbs, and Mr. Eric Helser. Each of you has, in one way or another, helped me better understand modeling and simulation and/or lent support to my efforts in culinary self-education. Even more, I have learned an immense amount about the world at large that no news media could ever provide. I hope to run into many of you further down the road.

This material is based upon work supported by the National Science Foundation (Grant No. BCS-0410269), and support through travel grants provided by ASU Graduate College and the Graduate & Professional Student Association. My thanks goes to all of them.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Hybrid simulation model application requires effective simulation techniques that are built upon sound modeling concepts and practical methods. The term "hybrid" describes a system of two or more sub-systems that are disparate. The disparity between the sub-systems may take a number of different forms such as timing or structural incompatibilities. This concept of hybrid extends to models of systems. Instead of abstracting away the major differences between disparate sub-systems, as would be done for creating a monolithic model, a sub-system model is devised that best represents each sub-system. Using this approach, the sub-system models themselves are likely to be disparate, and the resultant system model a hybrid. Disparities between the sub-system models must be accounted for and managed in order for the hybrid model to yield meaningful results. Knowledge of the model specification and simulation execution engine, and an understanding of the role that the domain plays within each, allows a composition approach to be properly devised. Using the appropriate composition approach can help mitigate the complexity of a hybrid model.

1.1 Systems with Complexity

An increase in computational resources has enabled modelers to incorporate more details into their abstractions of systems, adding to the model complexity (Booch et al. 2007). A common approach to managing a complex model is to decompose the whole model into smaller parts (*e.g.*, Eker et al. [2003]; Karsai et al. [2004]; Sarjoughian [2006]). The idea is that smaller model systems, with very specific sets of input, output, and dynamics, are easier to conceptualize and represent within a single model. Thus, simulation models have become an interrelated set of sub-system models as more and more details of the original system are added. The sub-system models are often devised and developed as individual pieces. As a result, a modeler may find that that each sub-system models is best represented by models

with different characteristics. Examples of these differences include timing differences (*e.g.*, continuous versus discrete-time), structural disparities (*e.g.*, hierarchical components versus a network of systems), and domain-specific implementations (*e.g.*, spatial resolution). The use of different models for sub-system models, while beneficial for managing the inherent system complexity, adds a complexity of its own as these different models must now be integrated to recreate the dynamics of the original, whole system (Davis and Anderson 2004; Manson 2001). Depending on the complexity of the original system, this may be a daunting task.

The term "complexity" characterizes one or more parts of a real system, or model of a system, as being difficult. This difficulty may manifest itself across one or more aspects of the system, such as the ability to understand the system, describe it, or predict its behavior. All software systems with a rich set of behaviors exhibit complexity — this includes simulatable models (Booch et al. 2007). The authors also state that this complexity *can not* be negated. It *can* be managed, but only if it is first understood. Software is inherently complex due to the complexity of the problem domain; the difficulty in managing the software development process; the arbitrariness of abstractions and numerous implementations that can be made about a system; and the ability to describe the systems behavior in software. It is stated by Booch et al. (2007) that this software system complexity exhibits itself with five common attributes. These attributes are a hierarchic structure, ambiguity of abstraction dependant upon each modeler, a separation between interactions within and between components, patterns, and underlying simple systems. There is a sixth attribute, heterogeneity of software pieces (H. S. Sarjoughian, pers. comm.). In summary, for modeled systems, complexity can stem from the modeling approaches and from the software toolkits used to develop the model system.

Another categorization of the complexity, given from a social and natural sciences modeling approach perspective, employs the terms *algorithmic complexity*, *deterministic complexity*, and *aggregate complexity* (Manson 2001). Algorithmic complexity refers to the difficulty in solving a problem mathematically, as well as simplifying that solution (a domain-dependent endeavor). Deterministic complexity involves the ability to determine the outcome of the model system given changes in initial conditions and input (*i.e.*, "Is the model deterministic?"). These two kinds of complexity comprise a holistic view of a model. In contrast, aggregate complexity emphasizes the importance of synergy arising from the interactions among a system's sub-systems (*e.g.*, emergent behavior). These tie in well with the concepts presented in the previous paragraph with the exception that the process for its software implementation is unaccounted. Software complexity contributors (*i.e.*, discreteness of model components, model component design, management of model development process, and the flexibility of model components to evolve) underlie all three complexity categories while developing hybrid system models within a specific toolkit. As stated previously, complexity may manifest across many facets of a system, and understanding the complexity is the first step toward managing it within simulation models.

1.2 Simulation Model Frameworks

As simulation models can be used to represent many different kinds of systems, there is a very broad spectrum of modeling and simulation applications. The decentralized communities of researchers and scientists who use modeling and simulation have responded in a number of ways to combat the growing complexity of their models. For example, some have adopted software-centric frameworks, while others have adopted theory-centric frameworks. The former is closely tied to a model development environment in which the criteria and use of models is constrained by the programming languages provided within. Typically, these

model development tools focus on a specific application domain. This benefits modelers conducting research in these specific areas as the tools use familiar language and concepts. However, this then may limit the types of systems that may be richly described with the development tool.

Theory-centric solutions take a broader approach using mathematical specifications that describe the model system. This specification includes both the structure of the model and the dynamics that enable the model to change state. For systems that are meant to interact with other systems, the specification also includes a formal description of the interaction from the system's perspective. Ideally in the theory-centric approach, the theory is decoupled from the implementation, allowing alternative frameworks to be developed to the specification. This then provides the modeler with the opportunity to choose a tool for the model development and simulation. Even if two models are developed using different tools, they maintain a connection through their theoretical foundation. The downside is that in maintaining generality, the languages and concepts used within the development tool may be difficult to grasp by those without the knowledge of the theoretical basis and the mechanics employed within the framework to conform to the theory.

The intent is not to imply that there is no theory behind the models being described as "software-centric". Rather, it is stating that the theoretical aspect is not being applied directly to the model specification, including the overarching dynamics[1] between model components. For example, a Java program can be written that simulates the workflow in an office space based upon a theory of workload and employee behavior. The object attributes and behaviors have no pre-defined structure that directly relates to any theory nor is there

[1]For clarity, throughout the remainder of this dissertation, the term "dynamic" will be used to describe activity from a generic model-perspective, such as state change. The term "behavior" is used to describe simulated activity based upon implemented domain knowledge.

a predetermined relation between any of the objects. The objects could employ any object behavior and interaction within the confines of the Java language. At most, the model uses best practices such as the Unified Modeling Language (UML)[2] and design patterns. If the modeler's theory of workload and employee behavior changes, an entirely different model may be developed. This second model could have absolutely no resemblance to the first, except that it is also in the Java programming language and is modeling workflow. The model structure, interface, interactions between components, attributes, and behavior could all be different. In essence, the system model approach is *ad hoc*. Alternatively, that same model could be developed as a Petri Net. A Petri Net is a mathematical modeling language for describing discrete distributed systems (Severance 2001). It too can be implemented using the Java programming language. However, most of the objects will have predefined purpose, attributes, and methods that relate to the theory behind Petri Net structure and dynamics. The theory behind Petri Nets defines how these objects interact concurrently in order to represent a valid Petri Net. In addition to these properties, the objects would also have other attributes and methods related specifically to the workflow theory under test. If the modeler's theory of workload and employee behavior changes when using Petri Nets, only the workflow-specific properties of the model change. Often, the theory used to define these model systems is drawn from well-defined studies of system dynamics (Severance 2001). Thus, while the behavior of specific components may change to suit the workflow theory under test, the structure of the model and how its components interrelate will remain consistent with the Petri Net specification.

[2]Though UML 2 provides methods to describe model dynamics; it does not define specific time-based dynamics and structure (see Pilone and Pitman [2005]).

1.3 Hybrid Model Simulation

Important aspects of creating effectual hybrid simulation models include having the ability to precisely formulate complex system model structure and dynamics, being capable of evaluating model system dynamics, and having the capability to clearly describe the validity of the model behaviors (*i.e.*, simulation results). This begins with the robustness of the data. Lack of data can lead to model ambiguity in both multi-model structure and behavior. However, data accessibility is not likely to be within the modeler's control. The ability of the modeler to define models that can be logically verified, and to employ execution engines for these models that can be validated, is the next key concept in creating creating effectual simulation models. The modeler both ensures that the model behavior is correct, and ensures that the execution engine which provides the model's dynamics does so in a manner that does not cause errors in the state of the model or its output. Model definition is the aspect over which the modeler has the most control through choice of system abstraction and model implementation. These choices lay the foundation for the types of model complexity that must be accounted for and managed within the simulation. The third key aspect of capable simulation models is the system verification and validation of the simulation results. Achieving valid simulation results, which is the ultimate goal of modeling and simulation, is supported by defining model system requirements within the confines of the domain and available data, building models that properly abstract the domain and that can be verified against the model system requirements, and validating the simulated model behavior. It is most heavily influenced by model definition. While the modeler has some control over how verification and validation (V&V) is achieved, V&V is made more difficult by the complexity factors within the models and how they exhibit themselves during simulation (*e.g.*, emergent behavior).

Using a theoretic approach to modeling and simulation improves the modeler's ability to ensure model correctness. Some general-purpose modeling and simulation frameworks and tools have mathematical underpinnings (*e.g.*, the Petri Net, described above). These have proven mathematical specifications that provides a model with a foundation that has already been shown to be correct in its structure and dynamics. The use of a specification reduces complexity by removing the arbitrary approach to abstractions and implementations that can be made about a systems structure and dynamics (refer back to section 1.1). Furthermore, an execution engine can be devised and/or examined to ensure that it complies with the described model specification dynamics. Using this specification allows the modeler to focus on domain-specific behaviors of the model instead of their general structural and dynamic attributes. Referring back to the Petri Net example, this means that the modeler can concentrate on ensuring that the workflow process behaviors are correct, and need not focus on whether the overall structure of the model and its approach to state change is correct. *Ad hoc* model development approaches (*e.g.*, those chiefly developed in terms of programming languages and software tools) can be used, but an additional burden rests with the modeler to demonstrate that the model structure and dynamics, and the execution engine's treatment of the model, are all correct. This is because there is no single specification against which a model can be validated. Furthermore, due to lack of model specification, there is no bounds to the algorithms that may be used to implement model dynamics. One modeler may validate a model in the framework but, because of the lack of rigor in how a model may be defined, structured, caused to exhibit dynamics, and interact; the validation is unlikely to hold for a second model developed in the same framework. Again, this is referring to the overarching concept of model structure and dynamics, not

to domain-specific behaviors for which both approaches (theoretical and software-centric) require validation.

Domain-specific frameworks and tools often implement libraries of behaviors to further reduce the modeler's development and/or validation workload. These libraries may be applied to both theoretical and software-centric frameworks. Obviously, the validation of the library can be only as good as the the default framework itself. A downside to domain-specific libraries is a more narrow application of the framework or tool. For example, specific types of workflows may be implemented as a library within a Petri Net development framework, and their specific methods may be validated with respect to that framework and a specific domain application. If an application of the framework is required that lies outside of the proffered libraries, then modeler must start anew and validate the devised models. This may prove even more difficult if the dynamics of the core framework are not easily surmised (Gibbs 2009).

This dissertation is not stating that *ad hoc* approaches to devising simulation models are of no value. What is being claimed is that if a researcher requires a good measure of confidence in the resultant simulation data, correctness that can be traced from data input to simulation output, then using a theoretically-defined model development framework offers distinct advantages over software-centric counterparts. The ability to verify model behavior and validate results is an easier task when the model environment and execution engine themselves are not under scrutiny at the same time. Theoretically-based development frameworks reduce the workload on the modeler because aspects of the entire model have already been shown to be correct. Furthermore, there is an explicit definition of what the model structure, dynamics, and interface entails, making it possible to very specifically compare and contrast disparate theoretical frameworks, and highlight the obstacles that

must be overcome in order to allow the models to be composed into a cohesive hybrid model of a single system.

With a formal specification of each model and the model's executor, a modeler is able to choose an appropriate composition method for the hybrid model. Each approach has it strengths and weakness in areas such as ease of applicability, model detailing, defining rich model interactions, complexity management, and model reuse. A taxonomy of the approaches is given in section 2.3.

1.4 Problem Description

Hybrid modeling and simulation provides substantial benefits in terms of defining how a model is designed and implemented to represent the system under study. However, as explained in the previous section, the composition of the disparate sub-system models creates its own complexity that must be managed. The intricacies of the problems created by the compositions are more apt to be unseen by those who approach the interaction from a purely data and control perspective. But, just as it is unnecessary for every programmer to learn the intricacies of a programming language, every modeler should not have to become an expert in modeling and simulation.

An approach is required that helps to manage the interaction complexity of models composed into a hybrid model. This approach should be accessible to a broad range of researchers and scientists who wish to take advantage of hybrid models. The approach should also provide some level of generality that supports reuse of modeling efforts. Last, but not least, the hybrid modeling approach should ease the difficulty of model validation and verification of the simulation. Thus, the modeler can gain confidence in the simulation results.

The very broad range of research topics and model specifications prohibits there being a single solution that can do everything — there is no silver bullet, as the expression goes. However, there is a concept of how a modeler may approach the aforementioned tasks. The concept is called the Knowledge Interchange Broker (KIB).

The KIB concept addresses the above issues by defining an intermediary model that handles the interaction between the composed models (Sarjoughian 2006). The KIB recognizes two aspects of every model. The first defines the model specification, the second defines the model implementation. KIB accounts for disparities in both of these aspects using a third model, an *interaction model*. The Interaction Model (IM) itself has a formal specification of structure and dynamics and a specific implementation of its specification. A most unique approach of the KIB is that the domain is accounted for and, indeed, drives how the interactions are defined and managed with the IM. The result is the fact that while the KIB concept is broadly applicable across many systems and many more domains, a unique IM must be created for each set of composed model specifications (see Godding et al. [2004]; Huang [2008]; Mayer and Sarjoughian [2007]; Sarjoughian and Plummer [(2002)]). Furthermore, while the theoretical approach to handling the interaction between different specifications may remain the same, the implementation of that approach may change significantly due to the domain under study. Thus, the specific attributes and dynamics of the IM, for a specific set of composed system model specifications, may change as required by domain. Therefore, while the KIB concept describes what should be done, it does not describe how — a significant undertaking for each set of model specifications. A challenge then arises in choosing model specifications and a domain that will provide significant benefit to a sizable number of researchers when light is shed upon a composition approach using the KIB concept.

It is important to note that KIB requires an explicit specification for every model. Thus, only theoretically-based models may be considered for composition. This should make sense because if an *ad hoc* model does not define model concepts such as time, an explicit structure, and explicit input/output interface, then these concepts can not be accounted for in the composition. What remains is data and control. By not accounting for the other model concepts, the modeler can not as adequately address the impact of one model upon the other due to the interaction. For example, one model may directly change the state of another in such a way that invalidates the data in the second model. The problem is not necessarily the fact that one model directly modifies the state of another. It is more having a clear description of how one model operates and within what bounds that is necessary to understand whether or not the first model's actions will invalidate the second. Contrarily, specifications of models have been demonstrated and/or proven to operate correctly within a set of operational bounds. With this knowledge, a modeler can define a formal interaction that works correctly within the confines of the composed models' specifications.

1.4.1 Model specification types

There is a growing population of modelers who use agents in their simulations, and whose work may benefit from a formal approach to incorporating agents into hybrid models (*e.g.*, Alessa et al. [2006]). While the definition of "agent" varies from researcher to researcher, the agent's role may be quite significant in the domain under study. However, few model specifications model both agent and another system quite well. Thus, to develop models with very detailed information about an agent, the agent's environment, and the interactions between the two, a general approach is desired. The approach should provide the complexity management discussed above, and allow a composed model to be modified with minimal impact to the other.

An agent may be defined in many different ways and one must be chosen that fits the needs of the KIB concept. A typical approach is to use a rules-based agent, one in which the agent's rules are defined by a set of if-then-else statements. There are numerous frameworks that allow the implementation of such behavior but whose specifications either are not specified clearly enough for use in a KIB implementation, or whose approach narrows the application of the agent (*e.g.*, it may be too domain specific). An alternative to focusing on the behavior is to examine the timing of an agent model's dynamics. Most agent models are discrete time-based systems. Thus, it would be beneficial to use a system specification such as Discrete Event System (DEVS) (Zeigler et al. 2000) to describe the agent. The reason is that DEVS can fully describe the agent as either a discrete-event or discrete-time system, but the specifics of the agent behaviors are unspecified, providing more generality in application of the approach.

Similar to the choice of agent model, an environment model must also have a clearly defined system specification and executor. Cellular automata (CA) (Ilachinski 2001) systems serve adequately as they can easily describe an environment in which a modeler may imagine an agent exists. Furthermore, there are numerous CA specifications with which numerous researchers are familiar. Additionally, CA specifications typically only define the structure and dynamics of the system, and interaction between system components (*i.e.*, individual automaton). Specific behaviors of the components are not typically defined by the specifications, further supporting generality of application.

Both the DEVS and CA specifications have a broad applicability to many different domains. DEVS is a component-based specification capable of describing continuous and discrete-time models. The models have an innate sense of time and a defined structure for component interoperability. CA are often used because a collection of simple rules can

simulate complex system behaviors amongst a group of independent systems with an inherent spatial relation (Anderson 2006; Ilachinski 2001; Wolfram 2002). However, existing CA specifications either describe the cellular network as a closed system (Anderson 2006; Ilachinski 2001; Wolfram 2002) or they are defined in such a manner as to allow integration of the CA with the same class of systems (*e.g.*, Muzy et al. [2004]; Ntaimo et al. [2004]; Wainer and Giambiasi [1997]; Zeigler et al. [2000]). As a consequence, these system specifications do not explicitly account for input and output (I/O) from the CA. However, for a model composition to be theoretically grounded, both of the sub-system model formalisms must possess formal descriptions of the I/O from their respective systems. Therefore, it is necessary to devise a formal CA specification that explicitly describes these I/O.

1.4.2 Model implementations

A hybrid model composition accounts for more than just the theoretical aspects of the models. It is highly domain dependent. As such, a dissertation on the composition between a DEVS and CA model could not be completed in detail unless a specific implementation and domain are chosen. CA are often used to model environmental systems within, for example, social science, military, and industrial research. As such, it seems reasonable to use a CA environmental model as part of the exemplar hybrid model. DEVS discrete, component-based structure is well-suited to describing rules-based agents; themselves components operating in discrete-time. Thus, an agent-environment domain perspective will be used to guide the composition. DEVSJAVA — a simulation environment for parallel DEVS (see section 2.1.3.1 and Zeigler et al. [2000]) — and the Geographical Resources Analysis Support System (GRASS) (see section 2.1.2 and GRASS [2009]) — a development environment implementation of the Geographic Information System (GIS) capable of man-

aging large geospatially-related datasets — will be used to create the DEVS model and CA model, respectively.

1.4.3 Problem summary

To summarize, in an effort to define an implementation of the KIB concept that may be useful to a broad range of researchers and scientists, a composition defining the creation of a hybrid model of an agent sub-system model and an environment sub-system model will be developed. The agent model will be specified by DEVS and the environment model will be specified by a CA. The model specifications will be implemented in DEVSJAVA and GRASS, respectively. An interaction model must be devised that accounts for the disparities between these two model specifications and implementations. Furthermore, as existing CA specifications are not adequate for the purposes of composition, a CA specification will be expanded upon to create a specification that is adequate. Lastly, to demonstrate the implementation of an interaction model, and to demonstrate the varying levels of generality gained during a composition, an exemplar model will be provided.

1.5 Contributions

This dissertation proposes a novel approach to compose a discrete-event system model with a discrete-time, composable cellular automaton into a hybrid system model. The Knowledge Interchange Broker (KIB) concept is applied to this end. The composition is defined from a theoretical perspective to ensure correctness of the resultant hybrid system model. A unique interaction model (IM) is conceptualized, formulated, and developed for describing the interactions between DEVS and CA models. The IM is designed to provide support for managing the composition complexity of the hybrid system model.

The development and application of the hybrid model is also discussed. First, an exemplar agent-environment hybrid system model called aeScape is presented. The agent

model is developed using the DEVS specification within DEVSJAVA. The environment model is developed using the Composable Cellular Automata (CCA) specification (a product of this dissertation) and implemented in GRASS. The intent of this hybrid model is to provide evidence of the soundness and applicability of the theoretical composition. Next, the concepts for the interaction model connecting an agent and environment model are discussed in the context of the Mediterranean Landscape Dynamics (MedLand) project. MedLand is an international and interdisciplinary research project (MEDLAND 2009) with the goals of modeling long-term dynamics of human landuse and Mediterranean landscapes, and applying the knowledge gained to current landuse scenarios. It is also the environment from which first-hand knowledge of the domain was gained in order to better understand and formulate the theoretical concepts.

The main contributions of this dissertation are summarized in the list that follows:

I. Describe a hybrid system model composed of a parallel DEVS specification model and a large-scale CA system model (see section 1.5.1).

   A. Use poly-formalism composition, which is based upon the Knowledge Interchange Broker (KIB) concept.

   B. Describe an interaction model and the disparities between the DEVS and CA models for which it must account.

II. Devise a new CA model specification, the Composable Cellular Automata (CCA), that explicitly accounts for input and output from the cellular network (see section 1.5.2).

   A. Develop an approach to provide timing for the CCA system dynamics when the CCA is implemented within an untimed environment using a DEVS model.

III. Implement the hybrid system model in aeScape, an exemplar agent-environment framework (see section 1.5.3).

   A. Develop a conceptual mapping from a rich application domain to an abstract hybrid DEVS-CA model, with an associated interaction model.

IV. Employ concepts from the hybrid system model composition to create an agent model and an interaction model for the MedLand project (see section 1.5.3).

   A. Create a graphical user interface that provides a central point from which initial values for many of the hybrid system model parameters can be set.

1.5.1 Interaction Model contributions

As the IM is a critical part of the contribution of this dissertation, the following list is provided to highlight IM-specific contributions. It contains the items that the IM accounts for between the DEVS and CCA models:

- Model specification disparities
  - Correctness of interaction
    - Mappings (*e.g.*, data aggregation and disaggregation).
  - Structure
    - DEVS is a hierarchical, component-based system specification.
    - CCA is a network of systems.
  - Timing and control synchronization
    - DEVS is discrete-event
    - CCA is discrete-time

- Model implementation disparities
  - Development environment language
    - DEVSJAVA is in the Java language; program remains running for long durations, with memory structures that are persistent as long as the program is running.
    - GRASS is a collection of independent C language modules that execute and terminate; data persistence is through file operations.
  - Scalability
    - DEVSJAVA overhead grows proportionally to number of objects.
    - GRASS can manage multiple millions of cells logarithmically.
  - Provide an efficient way to support data analysis.
- Domain disparities
  - Resolution
    - DEVSJAVA agents may tessellate the land into 25 meter $\times$ 25 meter areas, while the GRASS environment model uses cells representing 5 meters $\times$ 5 meters.
  - Scale
    - A few hundred to a thousand DEVSJAVA agents may exist (many represented within one component), while several thousand to multiple millions of cells may exist in GRASS.

The development of an interaction model is a direct result of the unique composition of DEVS and the CCA models. The interaction model uses an existing system specification

to model the interactions within the hybrid system model. Part of the effort of this dissertation is in choosing a system specification that best represents the interactions given the two model systems to be composed. Then, as the two composed models are implemented, the IM implementation must also be carefully considered and devised to account for the composed models' behaviors within the context of the domain. In essence, the abstractions defined within the interaction model are developed by managing the disparities between the models to be composed.

1.5.2 Composable Cellular Automata specification

Another part of the effort of this dissertation includes the development of a composable cellular automata specification. This was necessary to overcome the shortcomings of the current CA specifications with respect to composability, as described above. The CCA provides for input and output from the cellular network structure to other models, whether cellular or non-cellular. As such, the composition of the hybrid system makes use of a CCA sub-system model, which is implemented in GRASS (GRASS 2009). Furthermore, as GRASS does not provide an innate sense of time to its models, a technique was devised by which a DEVSJAVA model would provide a timing mechanism for executing the the CCA state transitions and input/output functionality.

1.5.3 Exemplar models

The exemplar model is based on Sugarscape, an agent-environment model that was first presented in Epstein and Axtell (1996). The Sugarscape model has been extended to highlight the features of a hybrid model, the IM, and the CCA. The exemplar model, referred to as *aeScape*[3] represents a landscape environment on which sugar grows. Agents, called *sugar fiends* or just "fiends" for short, live within the environment and survive on the sugar.

[3] "agent-environment hybrid Sugarscape".

The fiends must eat the sugar to survive but, in doing so, change the environment and its ability to support sugar growth. Thus, the fiends must move around the environment in order to ensure their survival.

In a real-world application, the composition concepts described in this dissertation are being used in MedLand to develop a hybrid system model of composed human-environment sub-system models in an agropastoral domain. The human model uses an agent-based architecture developed in DEVSJAVA. The MedLand environment model is developed within GRASS. The purpose in using this composition approach in the MedLand project is to provide a prototype laboratory environment in which scientists can study human-environment interaction phenomena.

1.6 Organization

Having familiarized the reader with the main, general objectives and goals in this dissertation, the remaining chapters will detail the hybrid agent-environment modeling concepts, methodology, implementation, and demonstration. Chapter 2, describes current approaches to similar domain problems. It also provides a detailed discussion of the Discrete-Event System and Cellular Automata specifications. Chapter 3 provides a detailed discussion on the system specification of the CCA models. It also defines the composition from a system-theoretical perspective, and describes the compositions of the DEVS and CCA models, as well as the devised interaction model. The next chapter, chapter 4, provides an in-depth discussion on the creation of the two models in their respective development environments. Chapter 5 describes the simulation of the exemplar models. It also describes how the poly-formalism concept was applied within the MedLand project. The last chapter, chapter 6 provides a summary of the work contain herein and also discusses some of the future work to be conducted, based upon the results of this dissertation.

CHAPTER 2

BACKGROUND

This chapter provides a familiarization with some of the existing works that relate to this dissertation. It does so by first providing a categorization of these works from an agent-environment modeling perspective. This categorization is then followed by a discussion of theoretical concepts applicable to this dissertation. These concepts include CA, a categorization of multi-formalism approaches, and a description of the KIB concept.

2.1 Agent-Environment Modeling Categories

A system model constructed from multiple sub-system models is not a new concept. Many approaches have grown from within specific communities to support some combination of human-environment modeling using multiple sub-system models (Parker et al. 2003). A variety of established and popular toolkits have been examined during the course of this research in terms of their capabilities for hybrid agent and environment process models. The most applicable are divided into agent-centric, environment-centric, and theory-centric categories. The agent-centric and environment-centric development tools are primarily built by casting domain-specific knowledge and modeling concepts to programming languages; whereas theory-centric tools emphasize the use of general-purpose modeling formalisms with proven execution protocols. Some major approaches that belong to these categories are discussed from the perspective of hybrid human-environment modeling.

2.1.1 Agent-centric

Various toolkits specifically for agent modeling have been under development since the 1990's and, in recent years, adopted by researchers and practitioners in social and environmental sciences, computer networks, and military applications. Some of the more popular ones amongst agent-environment researchers are NetLogo (Tisue and Wilensky 2004), Repast (North et al. 2005; North et al. 2006), and Swarm (Minar et al. 1996). Each offers a

development framework with libraries focused on modeling agents. While these approaches vary in their features, such as levels of support for data visualization and graphical user interfaces, models in all three are developed based on the concept of an agent. These toolkits provide pre-built agent models which may be modified per application domain.

The interaction between the software components is managed through interfaces that are defined in terms of software concepts and techniques. This is because these toolkits are developed based on general object-oriented programming concepts and techniques that are implemented in specific programming languages such as Java. The organization and behavior of the agents and their relationships may be defined using the Unified Modeling Language (Booch et al. 2007). While agent modeling toolkits can be interfaced with external environment model tools such as GIS (see section 2.1.2), the resulting integrated tools suffer from two problems. The first is that there is no specification that explicitly describes model structure and the model's dynamics as a system. As a reminder, the term "dynamics" is not referring to domain-specific behavior such as how the agent makes a decision. Rather it is referring to things such as the characteristics of how a model undergoes state change in discrete time. The second problem is that the software-based systems are interfaced using simply data and control. Consideration for how two model specifications may interact can not be accounted for because the specification is not provided. Further, a lack of specification does not imply that data and control interaction is all that is required. For example, a set of variables may define the state of a model. A simple object method call is all that is required to change one or more of these variables. By doing such, the state of the model may be invalidated the next time it tries to modify its own state. Thus, the interaction and the resultant simulation results are now invalid. Having a specification does not automatically prevent this problem from happening. It does, however, identify to

the modeler explicitly what should and should not happen such that when a framework is developed, it can employ means to prevent such invalid operations, or allow a modeler to recognize where caution is necessitated.

The unspecified nature of these agent-centric frameworks puts more of the onus of model validation on the shoulders of the modeler. While there may be general guidelines and consensus of models might be built within these frameworks, there is no specification against which the model structure and dynamics may be compared. The validation task is made more cumbersome when expanded across an interface to a heterogeneous model. For these reasons, the aforementioned agent-centric frameworks can not be used as-is in a formal, hybrid model composition.

2.1.2 Environment-centric

Geographic Information Systems (GIS) are a popular software tool for modeling environment processes. The Geographic Resources Analysis Support System (GRASS) (GRASS 2009) is a well-known, open-source GIS (Neteler and Mitasova 2004) that includes a georeferenced data management system that supports examination and modification of large data sets (*i.e.*, raster and vector data files). The data is stored in a file called a *map* in either a raster or vector format. GRASS is comprised of independent modules that can act upon the maps (*i.e.*, alter data in the files). The modeler is able to define complex dynamics by executing one or more modules against one or more maps either manually or via the use of scripts which contain a specified execution sequence (Neteler and Mitasova 2004). It is widely used throughout government, research, and educational communities that deal with large amounts of geospatial data. Some environment-centric development environments, such as GRASS, have the capability to support efficient calculations on very large data sets. However, development environments like GRASS lack concepts for con-

necting with other systems beyond simple data exchange, and they are not well suited for modeling rules-based agents or any other component-based model. Though the latter may be partially due to the use of the C programming language for implementation, a GRASS model specification does not provide specific GRASS model dynamics or external system interface. Furthermore, the map algebra theory upon which GRASS is developed does not directly support the representation of structured systems such as cellular automata models (Mayer and Sarjoughian 2009).

Map algebra is a way of relating two or more raster map attributes that represent the same geographic space; a concept that was first defined in Tomlin (1990). The maps are tessellated into equivalent numbers of rows and columns and operators are used to generate new maps based upon the values in the originals. Included in the class of basic operators are arithmetic, relational, Boolean, combinatorial, logical, accumulative, and assignment operators. Some GIS implementations (such as GRASS) may also have high-order operators such as functions, verbal statements, programs, and iterations. Operators like *addition* are similar to matrix algebra — the same $(row, column)$ of each matrix is added to produce the result. However, map algebra maintains this one-to-one locational correspondence even for operators such as *multiplication*, which matrix algebra does not (DeMers 2002). As an example, consider a map that contains slope values for terrain. A second map contains values for the amount of loose soil in the same geographical area. Map algebra can create a revised soil map dependent upon the relationship of the existing maps under a heavy rainfall by using the slope and soil amount for each region in the tessellation. From a formalism perspective, map algebra constructs by themselves are inadequate to enforce a strict formal representation of the environment dynamics to be modeled. This is because the constructs allow an open-ended application of data manipulation in an *ad hoc* fashion. This allows

one modeler to apply a set of map algebra functions in one manner, and another modeler to apply a completely different set. Thus, map algebra is more like a function library in a programming language — it eases data manipulation and has very broad applicability. So, while map algebra itself may bee too general to be used as a specification, what this means is that, used judiciously, a subset of map algebra's capability can be applied to implement a specification within a GIS.

2.1.3 Theory-centric

Theory-centric (or theory-based) approaches have mathematical formalisms within which every domain-neutral model has known structural and dynamical properties. A proof of correctness can be applied to models specified with the formalism and to the models interfaced within. However, theory-based approaches are general in order to remain applicable to a broad array of domains. For example, as described previously, DEVS is a modeling framework well-suited for describing discrete-event systems. Models within DEVS have a hierarchical relationship. Cellular DEVS, an extension of the DEVS, uses regular tessellations to model cell structures as a collection of individual automaton (*e.g.*, Ntaimo et al. [2004]; Wainer [2006]). DEVS could be used to represent agents while Cellular-DEVS can represent the environment processes. These two formalisms are closely related and together they can support modeling agent-environment systems. The strength of this approach is its formalism with built-in coupling concepts and formulation for composing agent and environment models together. The use of this approach weakens as large-scale models must be developed. The environment model may encompass a very large data set — on the order of millions of cells. However, while the hierarchical structure of DEVS fits well for an agent model, the cost of managing each cell as an individual object can grow exponentially (Kincaid et al. 2006). While it could be argued that parallel implementation can

increase execution efficiency for large-scale cellular models (*e.g.*, Zeigler et al. [2000]), an environment such as GRASS is simpler and still more efficient for the type of environment process dynamics considered applicable to research projects such as MedLand. Given DEVS applicability for agent-based models, and its usage within this dissertation research, it is discussed in much more detail in section 2.1.3.1.

In the late 1990's, GRASS was integrated with DEVS-C++ to support simulation of complex adaptive systems containing geo-referenced movement of satellites, aircrafts, and ground assets (Hall 2005). This approach uses High Level Architecture (HLA) (IEEE 2000) where different simulations are integrated using the Object Model Template (OMT). However, HLA primarily supports simulation interoperability and lacks conceptual and concrete capability for composing models (*e.g.*, Sarjoughian and Zeigler [2000]; Davis and Anderson [2004]).

2.1.3.1 Discrete Event System (DEVS) formalism

DEVS is a mathematical specification that can describe discrete and continuous systems as discrete-event models. DEVS uses a hierarchical structure of components to describe the relationship between modeled system components. The original version of DEVS, referred to as classical DEVS, did not account for the occurrence of possible multiple state transitions occurring at the same time. Parallel DEVS, on the other hand, allows the modeler to specify model state transition dynamics if two or more external events are received simultaneously or if an external event should occur at the same time an internal state transition is scheduled (Zeigler et al. 2000). It is parallel DEVS that is used throughout this dissertation, and is referred to simply as 'DEVS'.

2.1.3.2 DEVS specification summary

There are two types of model components in DEVS — atomic and coupled. The atomic model may be considered the leaf node in the hierarchical tree. The atomic model has explicit state and state transition functions that define its behavior. A coupled model contains two or more other (atomic and/or coupled) models. A coupled model's behavior is more implicit. Its behavior results from the behavior of the underlying atomic models and the coupling defined between the models that it encapsulates. A coupled model does not have its own set of states, as an atomic model does.

A parallel DEVS atomic model component, M, is specified as:

$$M = \left\langle X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \right\rangle, \tag{2.1}$$

and a parallel DEVS coupled model component, N, is specified as:

$$N = \left\langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC \right\rangle. \tag{2.2}$$

Given only these two mathematical structures, a modeler would have to know if the DEVS sub-system model being composed is an atomic model or a coupled model. This could complicate the composition at the formalism level as one tuple emphasizes state transition while the other emphasizes couplings (see appendix A for element details). However, there are two additional points which simplify the matter. The first is that composition composes models via coupling. Even if a sub-system model does not formally encapsulate its data, the interaction model infused between the sub-system models ensures that one does not have direct access to the other. Second, the DEVS specification exhibits a feature called closure under coupling (Zeigler et al. 2000). Simply put, closure under coupling enables a modeler to treat a coupled model as a black box and, in doing so, that coupled model

can be specified using the DEVS parallel atomic model component specification (equation (2.1)).

2.1.3.3 DEVS specification execution

Every atomic model in a DEVS system model has a corresponding "simulator" component in the simulator. For every DEVS coupled model, there is a corresponding "coordinator" component that coordinates the execution of the models contained within the coupled model. At the root of the hierarchy is a "root coordinator" which initiates the simulation cycles. The root coordinator uses a global clock to schedule the execution of transition and output functions for atomic simulators. The coordinators handle the input and output via couplings. This simulator component hierarchy is used to execute DEVS models in continuous time, using a discrete-event mechanism. This means that state transitions times need not be known *a priori*. While some may be predetermined, they may be overridden or delayed by external events.

The DEVS simulator uses a set of messages to communicate events and pass input and output between its components. Each simulator assigned to an atomic model maintains the time of the last event to occur in the system model and the time that the next event will occur for its associated atomic model. The simulators pass the time of its next event up to its parent coordinator. This repeats until the root coordinator has all possible events. At that time, the root coordinator time-orders the events and increments the simulation time to the next, earliest scheduled event. The current simulation time is then passed down from the root coordinator to the individual simulators. The simulators update the last scheduled event time and, if the last scheduled event now equals the time of the next scheduled event for their specific component, the simulators execute the respective transition function and/or output function of the model component. The output from these events is sent as

input to the appropriate models and the simulators update their times of last event and next event. Since the simulator deals with events, a model can change state and output data in zero time by having its respective simulator schedule its next event for the current simulation time and passing this information back to the root coordinator.

2.1.3.4 DEVS frameworks

The DEVSJAVA simulation environment is a Java implementation of the parallel DEVS formalism. The development environment provides a library of common classes (most notably, templates for atomic and coupled models) for building models to the DEVS specification, a simulator that can execute the DEVS models, and a visualization tool (SimView) to examine the DEVS models' interactions and state changes as the simulation proceeds. As Java is an object-oriented language, DEVSJAVA is comprised of a hierarchy of Java components.

The CoSMoS (Component-based System Modeling and Simulation) simulation engine supports visual experimentation configuration and run-time data collection and observation (CoSMoS 2009; Sarjoughian and Elamvazhuthi 2009; Sarkar 2009). It is also developed in Java, but provides more functionality than its DEVSJAVA counterpart. The CoSMoS tool enables simulation-based system design processes with support for model verification and simulation validation. It also provides the capability to develop partial DEVS models[1] (Elamvazhuthi 2008). The models are simulated in the DEVS-Suite simulation environment (DEVS-Suite 2009; Kim et al. 2009). DEVS-Suite provides the simulation capabilities for CoSMoS. These capabilities are similar to what is found in DEVSJAVA, and adds additional features such as time-based trajectories and data collection (Kim 2008).

The Java Runtime Environment (JRE) creates a *virtual machine*, the "Java Virtual Machine" or JVM, in order to execute Java code. It is this common, generic JVM upon

[1]CoSMoS can create simulation code that conforms to the DEVS-Suite simulation engine. Specific behaviors and transitions must then be manually inserted.

which all Java code runs and is the reason for Java's "platform independence" or lack of compilation to a specific operating system and processor set. The JRE employs software specific to each machine, thereby bridging the gap between the JVM and the real platform beneath. Thus, through its Java implementation, the aforementioned DEVS frameworks enjoy platform independence.

2.2 Cellular Automata

Weisstein (2008) describes *cellular automata* (CA) as a "collection of ... cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are then applied iteratively for as many time steps as desired". CA are described from a slightly different viewpoint by Ilachinski (2001) as "a class of spatially and temporally discrete, deterministic mathematical systems characterized by local interaction and an inherently parallel form of evolution." What both are saying is that a cellular automata is a networked system of individual automaton (most often referred to as a "cell"). A cell interacts with other nearby cells and all cells in the network undergo state change at the same (discrete) time and in accordance with the same functions (or rules). Note that not all CA employ discrete-time dynamics (Zeigler et al. 2000) and some operate within vector space (Shiyuan and Deren 2004).

The set of cells that affect the state of a particular cell is referred to as that cell's *neighborhood*. Borrowing notation from graph theory, the ith neighborhood of a cell, c, is the set of all cells that lie at a radius i from c (Barile and Weisstein 2008; Ilachinski 2001). The neighborhood is often considered the cell itself and all immediately adjacent cells (Weisstein 2008; Wolfram 1983), implying a radius of 1. This is also called "nearest neighbor" (Weisstein 2008). Considering the system structure from the perspective of a network, it is possible to conceive many different neighborhood variations — hexagonal,

triangular, and octahedron networks, for example. A tetragon is the most common configuration, with a square being the result if the cell represents the same distance in both dimensions (Ilachinski 2001).

CA offer distinct benefits to modelers modeling natural system phenomena in which the system is comprised of many homogeneous sub-systems. First, the modeler is able to abstract the behavior of a single sub-system into a single cell. Second, the interaction between the sub-systems can be distinctly modeled through choice of network topology and specification of how neighboring sub-systems impact one another. From relatively simple behavior implementations and network topologies, complex system behaviors can be achieved. Finally, the way in which CA are tesselated lend themselves well to visualization and to mapping onto real-world systems (Wolfram 2002).

2.2.1 Multi-modeling with CA

Most current CA approaches handle input and output from the cellular network in a manner that is not conducive to formal composition. Many of the publications for these approaches discuss neither the particular CA specification used, nor do they implement input or output from the cellular network. The CA represents the entire system. The interest generated by the work is in the rule or rules applied and the resultant patterns. Any number of the publications found in El Yacoubi et al. (2006) and Sloot et al. (2004) may be examined as examples. In Cell-DEVS, the specific modeling formalism is specified. However, the cellular network is abstracted as a DEVS coupled model component, and every "cell" is a DEVS atomic model. All input to the coupled model are sent simultaneously to all of the cells within the network (Wainer 2006). A downside to this approach is scalability. As every cell is managed as a separate component, and every cell may have eight pairs ports — one

pair for each neighbor — the amount of computing resources that a Cell-DEVS network consumes can grow large rather quickly as the cell-space increases.

A tool called JCASim uses a Java implementation of CA that proposes to provide "coupling of different CA" that "can have different state sets and different spatial or temporal resolutions". However, it provides no theoretical grounds and all discussions regarding I/O occur at the application level, using Java remote method invocation, and without any proof of correctness of the individual models which are made to interact (Briesen and Weimar 2001).

A formal approach called Complex Automata (CxA) is proposed for coupling CA that have different scales (Hoekstra et al. 2008; Hoekstra et al. 2007). The aim of CxA is on composition of two or more CA with potentially differing spatial and temporal resolutions. There is no explicit input and output as the external input to one CA is the state of the other being coupled to it. The input affects the current cell through boundary conditions and collision operators. Additionally, time is treated implicitly in the update function. The intent of CxA is to create a system of CA that mimic the original, more coarse-grained CAs; not to provide the capability to allow an external system (both CA and non-CA) to provide input to a CA (Hoekstra et al. 2008). CxA appears to be the closest approach to a desired composable CA, but it still does not provide the necessary input/output concepts required for hybrid model composability.

There appear to be no clearly defined specifications for input and output from a CA that allow the CA to interact with non-CA using system-theoretical principles, and that provides the flexibility to clearly represent the domain within the interaction — whether that be input to a subset of the network cells, temporal disparities between the two composed systems, or differences in spatial resolution and representation — by same. For example, if

an agent model is composed with a CA, the agent should be capable of affecting a subset of the cells in the CA directly, without applying input to a boundary and having that *propagate* to the desired cell(s). From a domain perspective in which the CA represents a physical landscape, this propagation would make little sense. Furthermore, if a climatology model is composed with a CA, the climatology model should not have detailed knowledge of the cellular network. Using a domain-specific mapping protocol such as { "North", "Northeast", "East", "Southeast", ... } allows it to be more easily applied to real-world environmental processes. The CA resolution could be changed or the network configuration could even change from a tetragon to a hexagon with no change to the climatology model itself. The mapping, obviously, would have to be modified but this is often a much easier and less error-prone endeavor than revising the code within another system model.

2.3 Multi-formalism Model Composition Approaches

Modeling and simulation is used across many disciplines, and does not have a unique, central core from which to draw specifications or concepts that are applicable to all. As such, some terms have different meanings dependent upon the organization using the term and the context in which it is used. Therefore, before elaborating on a taxonomy of specific multi-formalism approaches, a definition of the terms *formalism* and *realization* are given as they apply within this dissertation.

A model's formalism is its *specification* and *execution*. These are the mathematical descriptions of the system structure and dynamics (discussed in chapter 1) and the machinery to execute the model, respectively. Depending on the model development framework, the execution engine may be called an *simulation engine*, a *solver*, or an *executor*. The specification and execution layers are not specific to any one model instantiation. Rather, they describe any model that conforms to the formalism. For example, equation (2.1) (see

section 2.1.3.2) is a specification for a DEVS atomic model. An abstract DEVS simulator would describe the execution of atomic models derived from this specification (see Zeigler et al. [2000] for simulator details).

A model's realization encompasses *software architecture* design and implementation. The software architecture layer refers to software design (*e.g.*, described in UML) that can be forward engineered to a specific programming language (*e.g.*, Java, C, Lisp). Design considerations can be as simple as converting an integer from one model into a double for another, or as complicated as specifying quality of service or handling synchronous versus asynchronous input and output. The other aspect of realization specifies the *domain implementation specific details*. This refers to those elements of a model implementation that are domain, rather than architecture, related. These are often quantifications of the model's abstraction to the real system. As examples, whether a modeled landscape represents a 10m × 10m plot of land or a 5m × 5m plot, and if an agent model represents 10 people or 100.

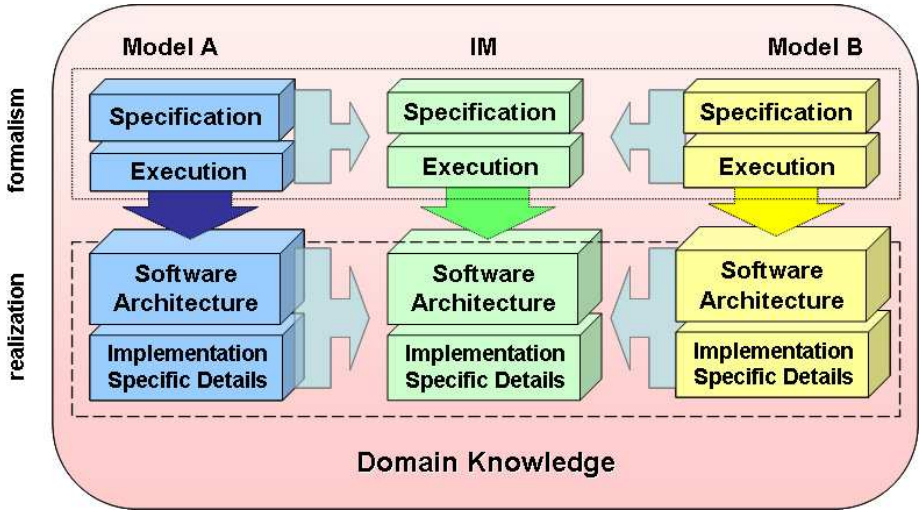

Fig. 2.1. Hybrid model formalism and realization aspects.

The formalism and realization aspects of a hybrid model are depicted graphically in figure 2.1. Details about the interaction model (IM) shown in the figure are provided in section 2.3.1.1. Note that in the figure, the domain knowledge encapsulates the models. The idea being conveyed is that the modeler's choices of formalisms and realizations, and an approach to composition of the models, is all driven by domain need (*e.g.*, data available, domain knowledge, and questions the simulation is intended to answer).

2.3.1 Approach taxonomy

A graphical depiction of the different approaches to multi-formalism modeling can be seen in figures 2.2 and 2.3. This particular taxonomy, which categorizes multi-model approaches based on the relationship between the model formalisms, was first introduced by Sarjoughian (2006). A summary description of each approach is provided below.

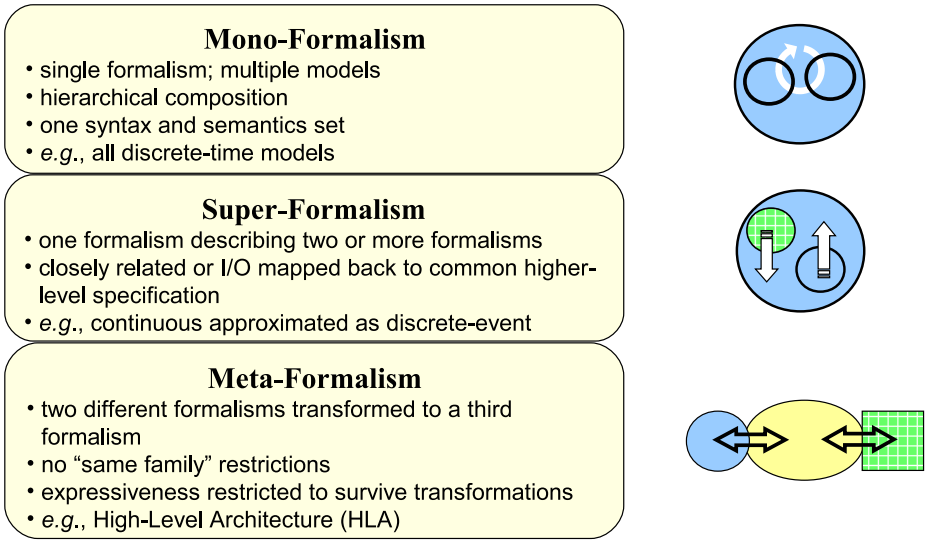

Fig. 2.2. Three approaches to multi-formalism composability.

A *mono-formalism* refers to one in which there is a hierarchical composition of models into/from parts described within one syntax and semantics. For instance, a system model composed of all discrete-time models.

The term *super-formalism* applies to a single formalism that supports describing two or more different types of models. It requires that the models be of the same family (*e.g.*, system specifications). Note that the super-formalism approach forces a uniform execution approach and syntax on both models. For example, the Discrete Event & Differential Equation System Specification (DEV&DESS) (Zeigler et al. 2000; Zeigler 2006) can describe both a continuous model and a discrete-time model.

A *meta-formalism* describes mapping two disparate formalisms to a third, common formalism. It does not have the same-family model restrictions that are levied upon the super-formalism. However, expressiveness of the model formalisms must often be restricted according to the metaformalism to ensure proper, multiple mappings. High-Level Architecture (HLA) (IEEE 2000) and Repast (North et al. 2006; North et al. 2005) are examples of this.

Poly-formalism model composition is an approach wherein models derived from disparate formalisms are composed using another model (see figure 2.3). This additional model succinctly models the data and control mechanisms between the composed models (Sarjoughian 2006). The poly-formalism model composition employed for this research is based upon the Knowledge Interchange Broker (KIB) concept of multi-model composability (Godding et al. 2004; Sarjoughian and Huang 2005). The KIB defines a composition between two models derived from disparate modeling formalisms. To compose the two models, the KIB accounts for differences in the formalism of each model, the realization of each model, and each model's domain-specific implementations. All three of these model disparities are accounted for within a third model, an *interaction model* (IM) (see figure 2.3) (Mayer and Sarjoughian 2007). The IM encapsulates the data and control mechanisms required for composition. The KIB concept does not specify which model types may be

composed. Rather, the choice of which two models to compose, and the domain in which they are applied, is used to guide the creation of an IM specific to that composition.

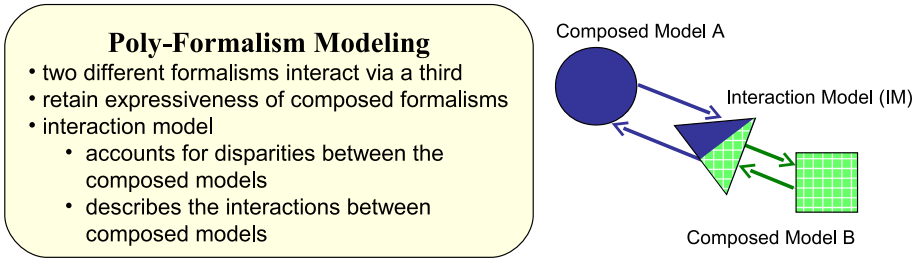

Fig. 2.3. Poly-formalism composability; a fourth multi-modeling approach.

Poly-formalism model composition removes the requirement for the composed models to have structural or dynamical knowledge of the other models. The composing model (*i.e.*, the IM in the case of the KIB approach) contains this knowledge for all of the composed models. This approach affords the composed models their independence and, as a result, provides the modeler with benefits. The first benefit is explicit interaction visibility and management. There is a central point at which all interactions occur and, therefore, at which the modeler need examine or modify what occurs. The first benefit lends itself to the second, an increase in model complexity management. By maintaining a central point at which the interaction is controlled, the complexity of that interaction can more easily be seen, understood, and managed than if it were distributed throughout the composed models. This includes direct input and output with the experimental frame. Furthermore, by removing the need for one or more composed models to have structural or dynamical knowledge of another, the complexity of each composed model's development is reduced. Each composed model may be designed, developed, and tested by domain experts with knowledge of only that sub-system, without having concern about how the model will interface with the other composed models beyond domain application.

2.3.1.1 Interaction model

The IM may use either of the composed models' formalisms or another one entirely; whatever best suits the needs of the desired interaction between the models for the given domain. Similarly, the IM is not restricted to any one of the composed models' execution methods, realization approaches, or domain-specific implementations. While the modeler is able to describe a general approach to devising an IM using only the system-theoretical aspects of the composed models, it is difficult to define the IM in its entirety due to the domain-dependent nature of the interaction between the composed models. For example, there are two models with discrete-time specifications. If the models are to be composed, and the interaction between the two models has a strong spatial relationship, then it may be appropriate to choose an IM that uses a discrete-time cellular automata specification. The IM devised from such a specification would be capable of synchronizing with the discrete-time (and therefore, known *a priori*) segments of each model. The CA structure and behavior would provide the spatial relationship between the two models.

The graphical depiction of the IM in figure 2.1 emphasizes that the formalism of the interaction model is chosen based upon the two composed model formalisms. From this formalism, the IM realization is created, again taking into consideration the requirements of making the two composed model realizations interact correctly. In all of the modeling decisions, the domain knowledge plays an important, foundational role.

2.3.2 Previous multi-formalism modeling approaches

Multi-formalism modeling is not a new concept. There are a few textbooks that specifically focus on multi-modeling approaches, including multi-formalism approaches (*e.g.*, Fishwick [1995]). The Department of Defense (DoD) actively pursues research into this area as well (see Davis and Anderson [2004]; IEEE [2000]). Additionally, a large number of research

papers have been produced on the topic from numerous, different sources (*e.g.*, de Lara et al. [2004]; Eker et al. [2003]; Huang [2008]; Godding et al. [2004]; Mosterman and Vangheluwe [2004]; Wainer and Giambiasi [1997]). Most of these can be considered "hybrid" models, as it is defined in chapter 1. There are approaches that use meta-modeling and others that use poly-formalism composition. Others focus on composition of the simulation instead of the models themselves. Many focus on DEVS, but there is also CA research as well. However, none focus specifically on poly-formalism composition of DEVS with CA that is developed from a non-DEVS formalism. Thus, while specific papers such as Godding et al. (2004) and Huang (2008) lend credence to the applicability of the poly-formalism composition approach using the KIB concept, both focus on the composition of a very different set of models and application domains. These differences express themselves as very different concepts for what is required to conceive, design, and implement an interaction model for agent-environment model systems.

CHAPTER 3

FRAMEWORK SYSTEM-THEORETIC FOUNDATION

In this chapter, the Composable Cellular Automata (CCA) is described from a theoretical perspective. With this description and an understanding of DEVS and poly-formalism described in chapter 2, the DEVS and CCA sub-systems can be examined and contrasted. There are a number of ways that a DEVS sub-system and a CCA system may differ, and these will be explained in more detail below. By exploring these differences with respect to poly-formalism composition, a domain-neutral understanding of the composition is gained. This provides a foundation from which to implement the hybrid system, including the interaction model, in its entirety.

3.1 Composable Cellular Automata Formalism

The CCA specification is based upon a multi-component Discrete Time System Specification (Zeigler et al. 2000). The CCA is a result of the work conducted as part of this dissertation (Mayer and Sarjoughian 2009). The CCA specification expands the standard cellular automata specification to explicitly include input and output from the cellular network. It manages this by treating the cell components as if they were encapsulated at the network layer. CCA mapping functions transfer network-level input to specific cell components. Cell input explicitly accounts for input received from its influencers and input received from outside the network, and may modify its state accordingly. Similarly, cell component's output is comprised of both output to the cells that it influences and output to the network-level. Furthermore, these two outputs need not be the same. A network-level mapping function maps these cell component output to a network output. Thus, the network-level presents an interface to other components and need not expose the details of its underlying cellular network structure.

As alluded to above, the cells themselves are components. This means that a cell's neighbors should not have direct access to a cell's state — not for viewing, and especially not for modification. Instead, the CCA specification is more of a push model in that each cell sends output to the cell's that it influences. The specific output that a cell sends to each influencee is determined in the output function. A cell could send all or part of its state. Alternatively, a cell could send some aggregate of the values that comprise its state or an arbitrary value. The same holds true for the value that the cell provides as part of the network output. This value may be a value that represents a subset of its state or a value that is determined arbitrarily.

3.1.1 CCA specification summary

A CCA is specified as a network of homogeneous components. There are two major equations that describe the CCA. A three-dimensional CCA model network, N_{CCA}, is formally described as:

$$N_{CCA} = \left\langle X_N, Y_N, D, \{M_{ijk}\}, T, F \right\rangle, \tag{3.1}$$

and each cell component, M_{ijk}, within the network is specified by:

$$M_{ijk} = \left\langle X_{ijk}, Y_{ijk}, Q_{ijk}, \mathrm{I}_{ijk}, \delta_{ijk}, \lambda_{ijk}, T \right\rangle. \tag{3.2}$$

The full CCA specification is provided in appendix B and more specific details about the elements are provided therein.

Equation (3.1) (above) details the CCA from a network perspective. A CCA network is defined as a sextuple of distinct sets. The first two, X_N and Y_N are the external input mapped to the network and external output mapped from the network, respectively. D is a set of indices that uniquely identify each component within the set of homogeneous components, $\{M_{ijk}\}$, that belongs to the network. T is a finite set of time-ordered, time

intervals that structures the discrete-time dynamics of the network. The last set, F, contains the mapping functions between the CCA components and the network as a whole.

Equation (3.2) defines a network component, M_{ijk}, with more detail. These components, or *cells*, are a septuplet of sets. The first two, X_{ijk} and Y_{ijk}, are the input to and output from each cell. Each is a union of data external and internal to the network. $X_{ijk} = \dot{X}_{ijk} \cup \overline{X}_{ijk}$, which represents input from the cell's influencers and external input mapped to this cell, respectively. $Y_{ijk} = \dot{Y}_{ijk} \cup \overline{Y}_{ijk}$, which represents output to the cells that this cell influences and output that acts as part of the external output from this network, respectively. Q_{ijk} is the set of possible states for the component. I_{ijk} is the set of indices that identify this cell's influencers (*i.e.*, its neighborhood). The component's state transition function is δ_{ijk}, and the output function is λ_{ijk}. T is the same set of time-ordered, time intervals that exists in the network tuple in equation (3.1). The significance of this is that it ensures that every cell in the network is using the same set of time-ordered, time intervals and, therefore, every cell undergoes state transition at the same discrete time. It should also be noted that a specific cell has no *a priori* knowledge of the network, including characteristics such as total number of cells in the network and connectivity (*e.g.*, Moore versus von Neumann networks).

While the network is the system of cells, for the purposes of better understanding the set of mapping functions, F, it may be easier to at first think of the network, N_{CCA}, as a shell around the set of cells, $\{M_{ijk}\}$, which comprise the system. Then, one can think of a mapping function which aggregates the individual cellular output, $\overline{Y}_{ijk}$, to the network output, Y_N. Similarly, another mapping function could provide the opposite, disaggregation of data from the network input, X_N, to the respective external input for each cell, $\overline{X}_{ijk}$. The functions f_{out} and f_{in} are the elements of set F from equation (3.1). They are the

aggregation and disaggregation functions that provide external output from and input to the cells, respectively.

As an example, assume that there is a two-dimensional CCA of dimension 3×3[1]. Using equation (3.1), this model's network may be specified by $\mathbb{N}$ for X_N, $\{0,1\}$ for Y_N, D as $\{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$, T as $\{1,3,1,1,2,1,2\}$, and the f_{out} and f_{in} mapping function set as

$\{\text{median}(\bigcup_{(i,j)\in D} \overline{Y}_{ij}) \mapsto Y_N,\ (X_N)\,\text{modulus}\ 2 \mapsto \overline{X}_{(2,2)}\}$, respectively. Each example cell, $M_{i,j}$, specified in accordance with equation (3.2), has $\{0,1\}$ for X_{ij}, $\{0,1\}$ for Y_{ij}, $\{0,1\}$ for Q_{ij}, $\{(i+1,j),(i-1,j),(i,j+1),(i,j-1)\}$ for I_{ij}, and T as $\{1,3,1,1,2,1,2\}$. δ_{ij} of M_{ij} is given as: if $q = 0$ and $\text{median}(\dot{x}, \overline{x}) = 1$, then $q' = 1$, else $q' = 0$. Finally, λ_{ij} is $\dot{y} = \overline{y} = q$. Here, in the previous equations, $\dot{x}, \overline{x} \in X_{ij}$ (where $\{\, \dot{x} = \dot{y}_\iota \,|\, \dot{y}_\iota \in \dot{Y}_\iota,\ \iota \in \mathrm{I}_{ij} \,\}$), $\dot{y}, \overline{y} \in Y_{ij}$, and $q, q' \in Q_{ij}$. Furthermore, q represents the state of M_{ij} at time t_r while q' represents the state at time t_{r+1}.

The exemplar CCA described above maps all external input to a single cell, $(2,2)$, and ensures that the input values are within the input boundary conditions required by component M_{ij}. Given this and the state transition function δ_{ij}, cell $(2,2)$ may undergo a state change that would not have been achieved if the input was dependent solely upon its influencers. This of course may have an influence on the states of its influencees in later time segments. Figure 3.1 demonstrates this.

In figure 3.1, t_0 is the time when the CCA is in its initial state, q^*, and we'll let that time be 0. Each subsequent time is calculated by adding the next element from T (*i.e.*, $\{1,3,1,\dots\}$) to the current time. Note that for the way that I_{ij} is defined in this example, M_{ij} is not one of its own influencers. It is only the cells directly above, below, and to the

[1] As the example is two-dimensional, subscript k is constant and, therefore, removed from the element identifiers.

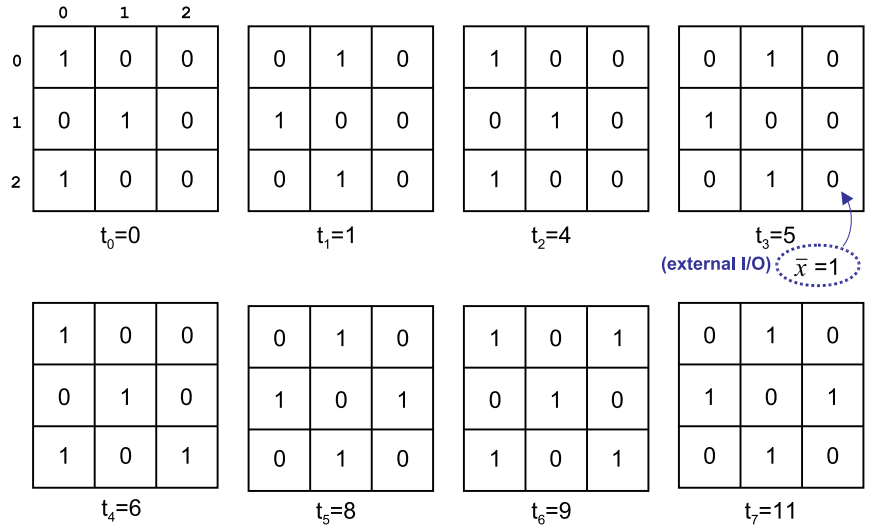

Fig. 3.1. Exemplar 3×3 CCA cell states at different time instances, t_r.

sides of the cell that influence its next state. For instance, the state of $M_{(0,1)}$ (first row, second column) is 0 at time t_0. This cell is influenced by cells $M_{(0,0)}$, $M_{(0,2)}$, and $M_{(1,1)}$, which output their current states at the end of the first discrete-time segment; 1, 0, and 1, respectively. There is no external input to this cell. When the state transition function executes at the start of time t_1, the median of these three input values are taken and the state of $M_{(0,1)}$ is set to 1 at time t_1.

A pattern is seen in the cell states from times t_0 - t_3 that repeats every two time steps. This pattern would repeat until the end of the simulation without any external intervention. However, at time t_3, external input with value 1 is mapped to cell $M_{(2,2)}$. This external input breaks the pattern by changing the state of $M_{(2,2)}$, as seen in figure 3.1, time t_4. This state change is similar to that which was described above for $M_{(0,1)}$ at time t_1. The state of $M_{(2,2)}$ at time t_4 is dependent upon the median of the input received from $M_{(1,2)}$ and $M_{(2,1)}$, and the external input, $\overline{x}$, received at the end of the t_3 discrete-time segment. The external input also, by chance, begins a new pattern that can be seen in

times t_5 - t_7, and would have been repeated if there were additional time elements within T. As the output from the network is the median of the output from each cell (which is simply the current state in the example), the network output Y_N would be 0 for all times except t_6. At this time, $Y_N = 1$. Note that this change in external output would not have occurred had the external input not been received.

3.1.2 CCA specification execution

The execution of a CCA model occurs in discrete-time. This means that the element T in equations (3.1) and (3.2) must contain a valid set of time-ordered, real numbers that specify when each discrete segment is to begin. Consequently, the length of each discrete segment can be determined from this time-ordered set. While every discrete segment need not be equal in duration, the length of each segment must be a value greater than 0 (per the definition of a discrete-time system). Input into a cell is received at the start of a segment. Also, a CCA cell may only change state at the start of a discrete segment. Furthermore, the input and the resultant cell state after a state transition function, are constant until the segment ends. As the cell may only generate output to its neighbors at the end of the segment, the output which occurs at the very end of one segment, becomes the input at the start of the next. It is this input which the cell receives and processes in its state transition function (see figure 3.2). The distinct separation of state transition occurring at the start of a segment and output occurring at the end prevents an infinite loop in zero time due to the bi-directional connectivity between cells within the cellular network (see Mayer and Sarjoughian [2009]).

At the start of each discrete time segment, all of the cells within the network update their current state in accordance with a transition rule that takes into account the states of each cell's influencers (Ilachinski 2001). In the context of equation (3.2), the transition

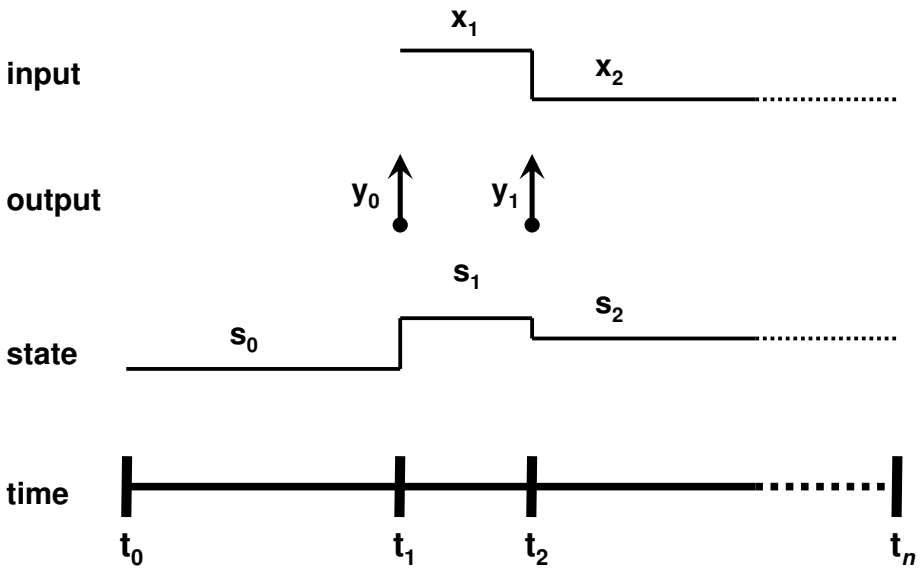

Fig. 3.2. Discrete segments example.

function, δ_{ijk}, takes into account the input, X_{ijk}, which includes both $\dot{X}_{ijk}$, the input received from its influencers, and $\overline{X}_{ijk}$, the input originating from an external system (*i.e.*, state data of the cell's influencers and of the external system). Since external input may be mapped to an arbitrary set of cells within the network, δ_{ijk} must account for the fact that $\overline{X}_{ijk}$ may be $\emptyset$. Thus, input must be guaranteed to come from the cell's influencers. As the cell components, M_{ijk}, do not have direct visibility of influencer state data, this mandates that a cell's output to the cells it influences, $\dot{Y}_{ijk}$, (and, therefore, the input of the influenced cells, $\dot{X}_{ijk}$) must not be $\emptyset$. Algorithm 3.1 provides a more formal description of a general algorithm for the execution of a model built to the CCA specification.

Consider that an external system will likely have a different structure than the CCA, may not provide external input to the entire CCA structure, may not have external input every execution cycle, and may not receive output from every cell in the network. It would be inefficient and, from a component visibility perspective, inappropriate, for the external system to map to every cell in the network. Thus, two other mapping functions are required.

Algorithm 3.1: General CCA Execution

Require: $T \neq \emptyset$ {T is the set of time intervals (see equation (B.1))}
BEGIN:
$r \leftarrow 0$ {current discrete segment}
$t_r \leftarrow t_0$ {current time equals initial time}
while $r < |T|$ **do**
  Retrieve h_r {the time interval, where $h_r \in T$}
  $t_{r+1} \leftarrow t_r + h_r$ {increment current time to end of segment}
  for all $(i, j, k) \in D$ **do**
    Execute λ_{ijk} {the output function of each component, M_{ijk}}
  Execute f_{out} {generate the external output from the network}
  Execute f_{in} {generate external input to each component, M_{ijk}}
  for all $(i, j, k) \in D$ **do**
    Execute δ_{ijk} {the state transition function of each component, M_{ijk}}
  $r \leftarrow r + 1$ {next segment}
:END

Let Θ represent the external modeled system and $G = \{g_{out}, g_{in}\}$[2] be the set of mapping functions that map external data coming *out* of N_{CCA} and going *into* N_{CCA}, respectively. Then, it may be stated that the output from the CCA network, N_{CCA}, mapped to Θ may be described by $g_{out} : Y_N \mapsto \Theta_{input}$. Also, the input from the external system mapped to the network may be described by $g_{in} : \Theta_{output} \mapsto X_N$. Function composition may then be used to define the entire mapping from the external system, Θ, to the cells, $\{M_{ijk}\}$, and back. Using the definitions of f_{in} and f_{out} from appendix B, $g_{out} \circ f_{out} : \bigcup_{(i,j,k)\in D} \overline{Y}_{ijk} \mapsto \Theta_{input}$ and $f_{in} \circ g_{in} : \Theta_{output} \mapsto \bigcup_{(i,j,k)\in D} \overline{X}_{ijk}$. G is not defined within the CCA specification as g_{out} and g_{in} operate external to the CCA network, N_{CCA}. G would be defined in a component that is external to the CCA. In a direct interface, within Θ itself. If poly-formalism is used, G would be defined within an interaction model (see figure 2.3). If feasible, $F \cup G$ is defined within the interaction model in order to capture the entire interaction between the models.

[2]The subscripts 'out' and 'in' are from the perspective of the CCA.

3.2 DEVS-CCA Hybrid Model

There are a number of key elements that must be highlighted in order to draw clear comparisons later on. First, note that the DEVS model is a component-based formalism which describes the model, M, as a single component. Second, the DEVS specification uses the concept of ports and values to define its input and output sets. The same value, v, on two different ports, p1 and p2, are not the same input. Third, there are three state transition functions. The first, δ_{ext}, specifies how the system responds to an external event. The use of the term event implies that the system does not have knowledge about the timing of the forthcoming external input *a priori*. The second transition function, δ_{int}, specifies the internally scheduled state transitions. In other words, the system transitions to a new state for a predetermined period of time. At the end of this time, the system undergoes a state transition to a new state for a new period of time. Note that, because the system is discrete-event, the amount of time spent in the state, as specified by the time advance function, ta, may be any positive real number ranging from 0 to $+\infty$. The third state transition function, δ_{con}, specifies what occurs if an external event occurs at the exact same time as a scheduled internal transition. Either the internal transition occurs before the external input is managed or vice versa.

An obvious distinction between the CCA and the atomic DEVS model is that, while DEVS describes a component (or a hierarchical set of components via closure under coupling), the CCA specification describes a network of components. However, similar to how DEVS closure under coupling allows the hierarchical system to be considered a single atomic model, the CCA specification is formulated to allow the input and output of the networked cells of the automaton to be treated as if from a single component source. Thus,

formal composition of a DEVS model and a CCA model need only consider composition between a DEVS atomic model (equation (2.1)) and a CCA network (equation (3.1)).

3.2.1 Composing DEVS and CCA formalisms

A key point to poly-formalism composition is that it does not require that one sub-system model have exposure to the underlying mechanisms of the other and, in fact, retards this exposure. Thus, formal composition of a DEVS model and a CCA model need not consider the entire DEVS hierarchy nor be concerned with the underlying CCA cells and how they are networked. Instead, the composition must only account for the atomic model specification presented in equation (2.1) and the CCA network presented in equation (3.1).

3.2.1.1 Input and output

One of the most obvious way to begin composing the DEVS and CCA formalism is through the input and output elements defined in their specifications. A DEVS model, M_{DEVS}, creates output, Y_M, as a port-value pair, (p, v). In contrast, the input to a CCA model network, X_N, is a collection of data that is mapped to the individual cells via mapping function, f_{in}. It may be assumed that this data takes the form of a cell reference-value pair, (c, v). However, not all of the cells may be explicitly referenced. This is because the data may contain a reference to a range of cells or f_{in} may map a single reference to multiple cells within the network. Furthermore, as some cells may not receive external input, the cell may not be referenced at all instead of explicitly stating that the data value is $\emptyset$.

As DEVS models are built in a hierarchical fashion, the point at which the DEVS model is made to interact with a CCA is at the highest coupled level. This means that there is only a single DEVS model that is interfaced with the CCA, and therefore the composition need not consider multiple DEVS model output being aggregated external to any one model in order to be provided as input to the CCA. However, it is possible for a

single DEVS model to have multiple output ports and to generate output across multiple ports simultaneously at a specific point in simulation time. Thus, parallel output from the same DEVS component is a concern. However, it is one that can be handled by g_{in}. Thus, a composition of a DEVS and CCA model must map $\{(p, v_{port})\} \rightarrow \{(c, v_{cell})\}$. Note that v_{port} may or may not be equal to v_{cell}, and $\left|\{(p, v_{port})\}\right|$ may or may not be equal to $\left|\{(c, v_{cell}\}\right|$.

As an example, consider a rainfall model described using DEVS. The climatology model is a DEVS coupled model composed of a sub-system that computes rainfall in arid regions and another that computes rainfall for regions with an excess of water. Each computes an annual rainfall value that is output from the DEVS rainfall model on two different ports — arid and lush. So, the set of output pairs may look something like, $\{$ (arid, 0.3), (lush, 2.1) $\}$. g_{in} then maps these two values to three — desert, rainforest, and forest, where the first two are the values of arid and lush, respectively, and forest is a weighted average of the two. Thus, the network external input data, X_N, would be $\{$ (desert, 0.3), (rainforest, 2.1), (forest, 1.0) $\}$, and this is also the data that is provided as input to the mapping function, f_{in}. At this point, we assume the labels desert, rainforest, and forest have meaning to f_{in} and that, from these labels, it can forward the data to the appropriate cells within an example 10×10 network. So, for instance, the cells may have labels identifying their types. The 30 desert, 10 rainforest, and 50 forest cells would each get the appropriate corresponding value for rainfall, while the 10 cells with the label tundra, would receive no external input.

A similar process would take place for CCA output, Y_N, being provided as DEVS model input, X_M. In this case, f_{out} and g_{out} would be used to map $\{(c, v_{cell})\} \rightarrow \{(p, v_{port})\}$. Continuing with the rainfall example, assume that the CCA provides an average evaporated water value, based upon water content of each cell. Then, each cell may output its water

content; one of many attributes that make up its state. The output mapping function, f_{out}, then averages these water values and applies some function to calculate an average evaporated water value. This value is output from f_{out} and, therefore, provided as input to g_{out}, which then applies the value as an external event on the correct port of the DEVS model.

3.2.1.2 Timing and synchronization

The second major requirement to composing the DEVS and CCA specifications is in managing their timing. The DEVS formalism is a discrete-event formalism that uses a time advance function, ta, and one of three state transition functions ($\delta_{int}, \delta_{ext}$ and δ_{con}) to change the state of the model. The times at which the model changes state is not known *a priori*. This is because while the models may schedule internal transitions at specific times; external events, which themselves are not known *a priori*, may override the normally scheduled internal transition. Models which undergo internal transition use δ_{int}. Models which undergo state transition due to external events use δ_{ext} and, in the event that an external event occurs at the same time as a schedule internal transition, δ_{con} is used. The resultant state of the model once it undergoes state transition using one of the three functions, given that it starts at a state s, is modeler dependent. It may or may not reach the same resultant state for each of the functions. Contrarily, the CCA formalism uses T, a set of known discrete-time segment durations, and δ_{ijk}, the state transition function for the cell component, to change state. The CCA model always undergoes state transition using the same function and using a set of times that are known *a priori*. Therefore, to compose the DEVS and CCA formalism, these timing differences must be taken into account.

The exchange of input and output between the DEVS and CCA models can be reexamined with respect to the timing mechanisms. As was previously stated, a CCA

may only receive input at the start of a discrete segment. Therefore, to compose the two formalism, the modeler has one of three choices: 1) only output from the DEVS model at the start of a CCA model's discrete-time segment, (2) allow the DEVS model to output at any given time but have the CCA model ignore any input that arrives out of synch with the start of its segment, or (3) allow the DEVS model to output at any given time but buffer the output if it is not in synch with a CCA discrete-time segment. Each of these approaches has its own consequences.

The first option, forcing the DEVS model to only output at the start of each CCA discrete time segment ensures synchronization of the state transitions of the two models but at a significant cost to the DEVS model. By coupling the DEVS model's output to the CCA segments, the DEVS model must be provided with explicit knowledge of the CCA models time segments *a priori*. This creates a tight coupling between the two models and requires that the DEVS model be changed if ever the CCA model's discrete-time segment lengths should change. Add to this the fact that in forcing the DEVS model to lock-step with the CCA, the modeler may hinder the ability for the DEVS model to represent a better abstraction of the original system being modeled.

The second option is to allow the DEVS model to generate output as it requires, but have the CCA ignore any input that does not come at the start of one of its discrete-time segments. This approach avoids the difficulties of requiring the DEVS model to have intimate knowledge of the CCA model. However, whether the CCA has discrete-time segments of all equal duration or segments of varied duration, it is possible that the output from the discrete-event DEVS model may never synchronize with the CCA model. To use this approach, the modeler must decide how the interaction will be handled with the lack of new data at the start of a discrete-time segment. Some possibilities are to simply provide no

external input to any cell that segment, continue using the last input, or to use a weighted average of input received in previous segments.

The third option buffers the DEVS model output that arrives before the start of a CCA discrete-time segment. The modeler must still decide what to do if a CCA discrete-time segment begins with no data, but it is much more likely that the previous data is less stale. However, in addition to deciding what to do if no data arrives, the modeler must also determine what happens if two or more DEVS model inputs arrive before the start of a CCA discrete-time segment. Some possibilities are to keep only the last or use a weighted average of all of the input.

The paragraphs above in this section speak of possibilities for managing the interaction between the DEVS and CCA models. Some of these may have seemed odd in the context of the two models interacting directly. However, this is where the application of an interaction model has benefit. If the modeler decides to buffer DEVS model data that is output to a CCA before the start of its discrete-time segment, it would have to be decided in which model this data should be stored and managed. If the interaction were directly between the DEVS and CCA model, one or both of the models would require some intimate knowledge of the other, perhaps again requiring the DEVS model to know the actual discrete times of the CCA, in order to make this work. However, by using poly-formalism composition, the mechanism to support this approach could be placed within the interaction model. It would have the detailed knowledge of both models to make it work, maintaining the flexibility and reusability of the composed models. The best approach can not, unfortunately, be decided by simply examining the two formalism. The creation of the interaction model is heavily dependent upon the domain in which the models are devised. The best approach of the three is no exception. The best approach to composing the disparate timing

between the DEVS and CCA models is dependant upon the domain. However, that being said, of the three approaches, the first approach is most opposite of what is trying to be achieved using poly-formalism composition.

3.2.2 Composing DEVS and CCA realizations

As stated in section 2.3, the realization of a model encompasses its software architecture and domain-specific implementation details. As both the DEVS and CCA formalism are domain-neutral, there are numerous approaches to implementing and executing their specifications. This dissertation employs DEVSJAVA and GRASS for the DEVS and CCA implementations, respectively. As such, all further discussions of model implementation will be from the perspective of these two software development environments.

3.2.2.1 DEVS realization environment: DEVSJAVA

As DEVSJAVA is a Java-language implementation, the DEVS models built within Java are object-oriented constructs. These models are a logical collection of classes and, as devised in following the DEVS specification, the term *component* can be used to refer to the whole DEVS model or a subset (such as an internal coupled or atomic model) with a specifically identifiable behavior and interface. Due to its hierarchical, object-oriented nature, models contained within a larger DEVS model have a *part-whole* relationship. For example, as shown in figure 3.3 if a DEVSJAVA model of a car is built, the car would be the overarching coupled model. Contained within car would be sub-system models for chassis, fuel, and electrical. These three components are the parts that make up the car model. The sub-systems, if they too are coupled models, may also be made up of parts. For instance, the chassis is comprised of the driveshaft and engine. The engine is itself made up of two cylinder models.

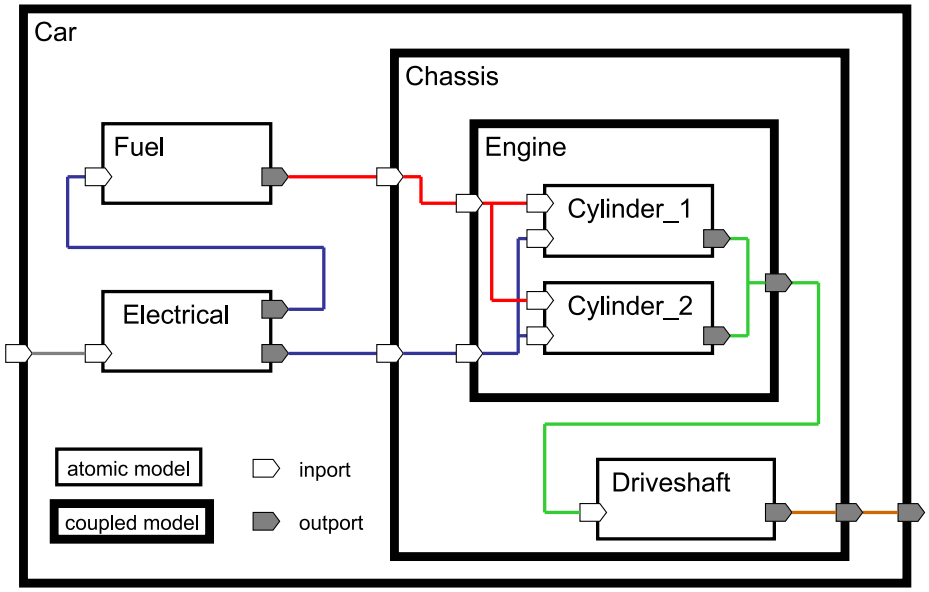

Fig. 3.3. DEVS model hierarchy example.

Communication up and down the DEVS model hierarchy occurs between atomic DEVS models through message objects. Coupled models have neither output functions nor external input functions and, therefore, do not send messages or act as the final destination for messages. A message is sent from one atomic model's output port to all of the model ports to which it is connected. Connections are restricted to either an output port of the atomic model's enclosing coupled model, or to input ports of other models that are enclosed by the same coupled model. Referring back to figure 3.3, an output port from the electrical model can either be connected to the fuel model, the chassis coupled model, or to an output port of the car model itself. If a message is sent to the enclosing coupled model's output port (*e.g.*, cylinder output port to engine output port), then it is immediately passed up the model hierarchy to the input ports to which that coupled model is connected (in this example, the driveshaft), or to the output port of a higher-level coupled model (*e.g.*, from the driveshaft to the chassis to the car). If the message is sent to neighboring model input ports then the message is either received as external input by an atomic model or is immediately

passed further down the model hierarchy when received by a coupled model (*e.g.* electrical model output to the fuel and chassis, respectively).

In an object-oriented frame, the output function of the model sending the message is called by the simulator and the message object is the resultant return value. The simulator then calls the external transition function of each receiving model and provides the message object as input to the function. In maintaining the hierarchical structure, messages can not be arbitrarily passed between two models that are not connected via ports. Thus, in models with a large hierarchical depth, messages may have to be passed up the hierarchy before they can be passed down another branch to reach the intended atomic model. For example, consider that the electrical model may be decomposed into battery and alternator sub-systems. An alternator uses the driveshaft's motion to generate electricity and recharge the battery. Thus, the driveshaft output (via message object) would be passed up to the chassis model, over to the electrical coupled model, and then down to the alternator model within. An alternative approach is to "flatten" the hierarchy by removing all coupled models except for the highest level (in this case, the car) (Zeigler et al. 2000). However, doing such looses some of the desired model qualities such as component-to-domain abstraction and component reusability.

In the simulator, each atomic model has a corresponding simulator component that stores the current simulation time and the time of the next event for its respective atomic model. This innate sense of time within a model gives chronographic meaning to its state data. Typically, the DEVS models do not store previous state data. The attributes of the DEVS model objects (*i.e.*, variables) are stored in memory. As the state transitions occur, the current variable data is overwritten in memory with the new data. This means that

any function which relies upon prior state data must employ a data structure specifically implemented to contain the necessary data.

3.2.2.2 CCA realization environment: GRASS

The composable CA specification can be rigorously implemented within the GRASS environment by being particular about which GRASS modules are employed and how they are applied against the maps. The CCA implementation (or realization) may then be used in formal hybrid model compositions, such as poly-formalism. While the CCA cells are not truly components, as represented within the GRASS environment, breaking this veil could lead to potential problems. For example, GRASS modules can be employed in a manner that map values (and therefore cell states) are modified without regard for the formal mechanics of the state transition function. However, if external sub-system models are allowed to do this, the potential exists that the values may be set to an invalid value that causes an error to propagate when the cell sends output to its neighbors. By ignoring the interface specified by the CCA specification, it becomes more difficult to manage the interaction between the systems and, therefore, the correctness of the model becomes suspect.

A CCA model developed in GRASS is comprised of the maps that store the cell state data; a specific subset of GRASS modules that perform the input, the output, and the state transition function execution dynamics; and a set of scripts which order the GRASS module execution to conform to the CCA specification. The attributes that a cell models comprise the state of a cell. For example, land cover within a CCA model may be abstracted to one value (*e.g.*, foliage density), or many values (*e.g.*, slope, foliage density, and soil depth). Each attribute of a CCA cell is stored within a separate GRASS map. The inherent geospatial relationship between the data ties them together as a whole through map algebra operations, using the GRASS modules. In the form of a map, "cells" are

not components and, therefore, do not communicate with one another. The influence of neighborhood state on a cell is generated through the use of neighborhood functions in map algebra.

Every simulation cycle, as the state of the cells change, the existing cell data in the maps is not updated. Instead new maps are created with revised data[3]. Output from influencer cells to the influenced cell is managed by querying the state data of the influencers, performing the appropriate state transition function on the data, and then writing the resultant state data to one or more new maps. Unless the modeler explicitly chooses to do such, the previous maps are not deleted. As a result, prior state data is readily available if needed. However, there is no implicit sense of simulation time within a GRASS model.

3.2.2.3 DEVSJAVA and GRASS disparities

The models realized in DEVS and GRASS are very different in terms of architecture and execution mechanisms. DEVSJAVA uses a hierarchy of components while GRASS is a set of flat-file data and execution modules. DEVSJAVA models remain active as a program between simulation cycles and, therefore, can make use of memory to store data. On the other hand, GRASS models are only active as programs while a specific script or module is being executed. Thus, persistent data must be stored in a file, usually a map. GRASS modules can access a specific cell attribute in a specific map directly while DEVS models must pass data up and down the model hierarchy via message objects. Lastly, DEVS models have an implicit sense of time while GRASS models do not. To properly compose the two models, these differences must be accounted for in their interaction.

[3]This describes typical high-level access. Low-level access to the maps can be made that modifies current map values directly.

3.2.2.4 Timing

The major issue in composing a DEVS model realized in DEVSJAVA and a CCA model realized in GRASS is timing. As the DEVSJAVA model has an innate sense of time, attention will be focused on the GRASS model. An external program must provide a sense of time to the GRASS CCA model since GRASS modules execute a single operation, and then terminate. One approach is to contrive a timing mechanism in a script. As the script is typically executed and, in turn, this executes the specific GRASS modules, this is a reasonable choice. However, for the purposes of composition, the DEVSJAVA model and the script mechanics would still require synchronization. A more relevant choice for this dissertation is to use a DEVSJAVA model.

A DEVSJAVA model's innate sense of timing can be taken advantage of by forcing the state transitions to occur when a CCA state transition function, input function, or output function must occur. The DEVS model can then execute specific scripts, which call the appropriate GRASS modules, to perform the CCA operations. Manipulating the state changes of a DEVS model to provide timing for a GRASS-implemented CCA requires some thought, but it is quite feasible. Furthermore, it provides the benefit of simulator synchronization between the DEVS model and the CCA by using the same simulation engine to execute both the composed DEVS model and the DEVS timing model for the CCA.

A combination of GRASS modules (organized and applied via scripts), GRASS raster maps, and DEVS timer atomic models is used to realize the composable CA formalism. The GRASS module `r.mapcalc` performs the majority of the GRASS map algebra operations. It may be applied to provide internal and external input, to emulate the transition functions, and to generate the output. It accepts one or more existing maps as input,

performs a set of operations on those maps, and produces one map from the result. Modules such as `r.stats`, `r.info`, and `r.what` are map data query functions and can be employed to obtain and format data as external output. They simply query all (`r.stats` and `r.info`) or specific cells of a map `r.what`) and provide the result as output. How these modules are employed may vary dependent upon domain application.

Following algorithm 3.1, the execution within GRASS and DEVS can be specified. To clarify the relationship between the state-based timer model and the composable CA, the timer model will be given two states — input and output. Also, in order for a timer model to change state, it must undergo either an internal or external transition (caused by a time-based, state-specified internal change or input from an external entity, respectively). As both the input and output states are necessary regardless of whether or not there is external input, this means that the timer model will undergo two internal transitions (from output to input, and then from input back to output) for every CCA discrete segment. Note that, in each discrete segment, the CCA executes both its output function and transition function (see equation (3.2)). In the list that follows, the timer model execution is described.

Let r be the current segment where $0 \leq r < |T|$ and let the timer model be initialized to the output state at the start of a discrete-time segment at time t_r. Then,

1. The timer model undergoes an internal transition to an input state at time $t_r + h_r$. Before entering the input state, the DEVS timer model output function is called. The output function calls the *output function script*, which is the CCA cells' λ_{ijk} in equation (3.2). This script queries all cells within each GRASS map that represents the composable cellular automaton and generates an output map based upon the state of each cell as represented by the map values. In GRASS terms, each cell (i, j) is read

in each map that represents an attribute of the CCA state, and creates an output matrix based upon those cell values. This matrix is stored in another GRASS map.

2. Also, before entering the input state, the *output mapping script* is called to collect the values from the output map and format it as representative output from the mapping function, f_{out}. This data is then sent out from the network. Once the f_{out} output data is sent, the timer model enters the input state.

3. If, and only if, the timer model is in the input state when external input arrives, the timer model calls the *input mapping script*, which is representative of f_{in}. This script converts the received data into an input map with the same regional specifications as the maps which comprise the CCA model. If the timer model is in the output state when the external input arrives, the input is ignored.

4. The timer model remains in the input state for zero-time (simulation clock) and immediately schedules an internal transition back to the output state with the time to remain in that output state equal to the next scheduled time interval, h_{r+1}. If external input is received (as described in the previous item), then it is handled before the internal transition is scheduled.

5. Before the timer model undergoes its internal transition to the output state, its output function is called. The timer model output function calls the *state transition function script*, which is the CCA cells' δ_{ijk} in equation (3.2). This script evaluates both the current CCA maps and the map of external input (if it exists) to create revised values for one or more cells in one or more maps, thus changing the state of the CCA.

6. The timer model enters the output state and, therefore, the next discrete-time segment.

The above algorithm represents one possible approach to handling the external input. This method enforces a discrete-time handling of input at the start of a timing segment, as discussed in section 3.2.1. The dynamics and the pieces which play a role in the algorithm described above can also be seen in table 3.1 and figure 3.4. Note that figure 3.4 is presented from the DEVS timing model's perspective. Therefore, there are two states — input and output. The CCA cells have at least two states and are not restricted to "input" and "output". Additionally, there are two DEVS internal transitions specified (before which, the output function of the DEVS model is called). However, the output from the CCA only occurs one time, in step 2 of the algorithm above (the DEVS output function from the output state, in figure 3.4). In figure 3.4, it should be noted that there are two DEVS transition functions from the output state to the input state. The function used is dependent upon whether or not external input is received. If external input is received, the DEVS confluent transition function is used. For this model, the state transition to the input state is handled first. Then, the external events (the input) is handled next. Thus, the timer model will be in the input state when the input is handled. For the transition from input state to output state, the DEVS model output function executes the CCA state transition function, δ_{ijk}. If there is no input, then just the internal transition function is executed and, as before, the output function is called, which executes δ_{ijk} in the CCA. Table 3.1 provides a step-by-step relationship between the algorithm above and figure 3.4. This separation of mechanics from domain dynamics representation should help enforce to the reader the concept that the DEVS atomic model mechanics are being manipulated to provide timing to a composable CA that is realized in GRASS.

TABLE 3.1. Parallel atomic model for timing and I/O relationship to CCA

Step	DEVS Timer Model	Script	Result
1	output function, (output state)	*output function script*, λ_{ijk}	CCA cell output map
2	output function, (output state)	*output mapping script*, f_{out}	CCA output, f_{out}
3	external transition function, (input state)	*input mapping script*, f_{in}	CCA external input map, $f_{in} \circ g_{in}$
4	internal transition, (input state)		
5	output function, (input state)	*state transition function script*, δ_{ijk}	revised map values (*i.e.*, cell state)
6	internal transition, (output state)		new discrete-time segment

3.2.2.5 Synchronization

As the DEVS and CCA models represent different aspects of a hybrid system, so too might their abstractions manifest as timing differences. As an example, consider a DEVS agent that changes its state and performs some action on the environment every time it undergoes an internal transition. Similarly, the CCA environment model changes its state and sends output to the agent model every discrete-time segment. In the most simplistic of timing synchronization cases, one unit of simulation time will pass for every agent internal transition, and one unit of simulation time will passes for every discrete-time segment in the CCA environment model. In this case, the timing for the two is synchronized. However, this need not be the case. For example, allow one cycle to represent a year within the domain being modeled. While it may make sense to keep the agent's actions representative of a yearly activity, the environment model may be better represented by specifying that its dynamics only occur once every decade (10 cycles). The DEVS agent model and CCA environment model are now out of synch.

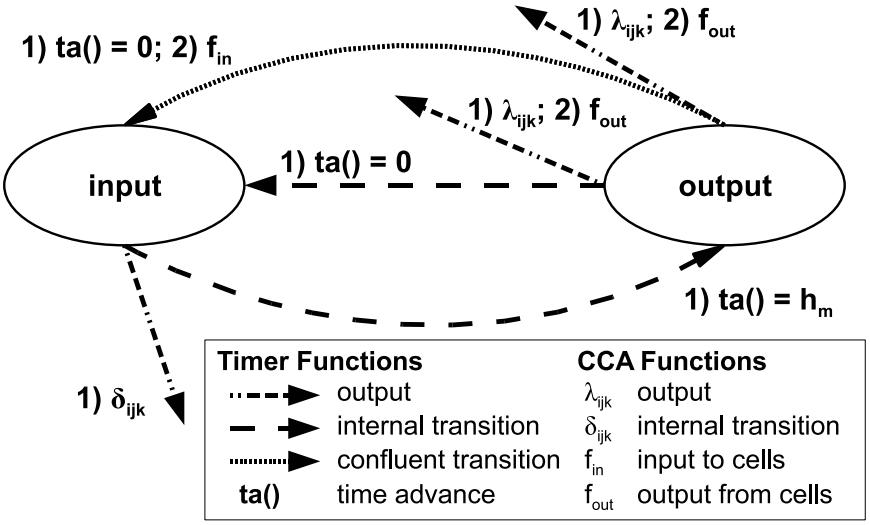

Fig. 3.4. Timer model statechart for CCA realization.

Timing synchronization is domain-dependent endeavor. In the example above, the approach to use to synchronize the agent and environment models depends on what their interaction means in the context of the domain. For example, if the environment state changes over the 10-cycle period are minimal, it may be feasible for the agent to continue to use the last value that was output from the environment model. Similarly, if the agent output has no long-term effect on the environment, the environment may not need to know what the agent did over the course of the ten cycles, except during the time at which it undergoes a state transition. However, if the two models are sensitive to the other model's activities, then a more complex synchronization is required. Thus, the environment model may need to know exactly where the agent was during the 10-cycle period. Similarly, the agent may require some representative change in the environment in order for its domain abstraction to make sense.

Timing synchronization becomes more of an issue when the modeler accounts for the fact that the DEVS agent is discrete-event while the CCA is discrete-time. This means that

the CCA environment will undergo state transition and generate output at predetermined times based upon each discrete-time segment. The DEVS agent model, on the other hand, may be time synchronized with the environment model for some period, and then fall out of synchronization upon receipt of an event. This event may change its state, generate output, and/or cause it to exhibit very different behavior. Given that the discrete-event CCA can only change state at the start of a discrete-time segment, the composition must account for the possible changes that the agent model may undergo, at times that do not correspond to the environment model's state changes and ability to accept external input.

3.2.2.6 Data mappings and exchange

The exchange of data between a DEVSJAVA and GRASS model requires a bi-directional mapping between a message object and a set of primitive values that can be used in a map algebra operation, including being stored in a map. Data input into a GRASS model can be parsed and managed directly by a script or GRASS module, thereby providing immediate use to the receiving map or maps representing the cell or set of cells. Alternatively, a GRASS map can be created external to the CCA model, which the CCA uses as input. However, a DEVS model must be devised such that the ports and couplings from the highest coupled model, down to the recipient atomic models, are structured such that the data can successfully traverse the hierarchy. The same is true of data going out from an atomic model to the CCA.

Data exchange between two models with different representations of the same domain requires the modeler to account for both domain differences and differences in how the individual models represent the domain knowledge. Differences can be as simple as changing an integer value to a real number. However, the differences between the two models can also be much more complex. DEVSJAVA models communicate via objects. GRASS maps

use primitive types such as integers and real numbers. A landscape model in GRASS may represent the landscape with a very high resolution, while agents may use a much lower resolution. For example, a landscape model specifies that a cell represents a 5m $\times$ 5m area. The agent model is developed such that each cell represents a 50m $\times$ 50m resolution. An approach needs to be devised to map from one model to the other, and then back again. With 100 landscape cells per agent cell, the 100 values could be averaged, and perhaps weighted.

3.2.3 DEVS-CCA interaction model

The DEVS-CCA interaction model accounts for the types of disparities that are described in the previous sections. To reiterate what has been emphasized previously in this dissertation, the development of the IM is very heavily domain driven. As a result, so too are the specifics of the IM design and implementation. Therefore, the description of the IM from a theoretical perspective must be kept general. More specific details about the IM are provided in the next section, which discusses implementation.

One aspect of the IM design decision that can be addressed immediately is timing. With a discrete-event agent model and a discrete-time CCA, an IM formalism needs to be chosen that is suitable to interact with both models. The IM-to-composed model interaction should be conducted in a manner that is indistinguishable to the composed model from dealing with its own class of models. Recall that the goal is to remove the composed models' knowledge of external model systems. From a timing perspective, using a discrete-event formalism enables the IM to interact with any discrete-time or discrete-event model. Using a discrete-event formalism, the IM can accept output from either model at any time, without having to know *a priori* when that output will arrive.

The IM is thus chosen to be described by the DEVS atomic model specification (equation (2.1)). When trying to understand the elements of the specification as they apply to the IM, it may be helpful to focus on the fact that the IM models the interactions themselves. The input set of the IM, X_{IM}, is the output generated by both the agent model and the CCA. Thus, $X_{IM} = Y_{Agent} \cup \overline{Y}_{CCA}$. Similarly, the IM output set, Y_{IM}, is the set of possible inputs to the two composed models, $Y_{IM} = X_{Agent} \cup \overline{X}_{CCA}$. The state of the IM represents the state of the interaction between the composed models. The set of possible states can vary widely depending upon domain need. In the most simplistic of cases, the interaction is a timeless event — consuming no simulation time and requiring no memory once complete. For this case, the IM may employ two phases as a DEVS model, passive and send, for when it waits for input and sends mapped output, respectively. More complex cases, such as when data persists or there is a delay between receipt of input and transmission of the mapped output, will require a more elaborate state scheme.

While not impossible, it is unlikely that the IM will have its own behavior that is not related to the support of the composed model interactions and their specific I/O. As such, most of the work for managing the interaction is likely to occur in the external transition function, δ_{ext}, where the input is received and the appropriate mapping functions are applied. δ_{int} is most likely to transition the IM back to a wait state for the next composed model input. Unless a more elaborate state scheme is devised, the default δ_{con} that specifies that an internal transition is to be conducted before handling any external events should suffice. The output function, λ, simply sends out the mapped data, or metadata, if the generated data is a large, externally-created data structure like a GRASS map. The IM can perform all mappings within zero simulation time or, if the composed models require, can arbitrarily set a delay before the mapped values are sent to the recipient model. While the

specification details are important, the critical part of the IM is its set of mapping functions, U.

The IM function set, U, maps the output from one composed model to the other. If the set of functions that map CCA output to agent input is defined as $V \equiv \overline{Y}_{CCA} \mapsto X_{Agent}$, and the set of mapping functions from agent output to CCA input is defined as $W \equiv Y_{Agent} \mapsto \overline{X}_{CCA}$. With $V = \{v_1, v_2, \ldots, v_m\}$ and $W = \{w_1, w_2, \ldots, w_n\}$, $U = V \cup W$, and $\{\forall u \in U | u \equiv y = f(x, s, t)\}$, where $x \in X_{IM}, y \in Y_{IM}, s \in S$, and $ta() \vdash t$ (see equation (2.1) for the definitions of X_{IM}, Y_{IM}, and S). In other words, for every input in the IM input set, there is at least one mapping function that will generate an output from that input.

In the case of an interaction model that composes a DEVS model and a CCA, the IM provides the mapping between the message object and primitive data types discussed above. Obviously, when input comes from the agent, the mapping function that is employed must transform the message object to the primitive data type and insert it into a GRASS map. Conversely, when data comes from the environment model, the transformation is from primitive type to object data structure. The DEVS concepts of ports can help facilitate the management of mapping functions. Providing the IM with an input port for the environment model (ip_e) and another for the agent model (ip_a) allows the mapping equations provided in the subsequent paragraph to be revised with more detail, using port-input pairs. Functions for environment-to-agent mapping are defined as $\{\forall v \in V | v = f(\{ip_e, x\}, s, t)\}$, and functions for agent-to-environment mapping are defined as $\{\forall w \in W | w = f(\{ip_a, x\}, s, t)\}$. The explicit usage of ports in managing the functions allows the modeler to focus on a subset of the mapping functions during development and test. However, as demonstrated in the

exemplar model in section 4.3, since all of the data is being managed from a central model, data from two different ports can be used within a single mapping.

One difficulty in developing a hybrid model is determining where each piece of the modeled system should be represented. In a GRASS-devised CCA, the DEVS timer provides timing control for the CCA functions. It might also appear to provide an interface to the script functions. As the timer model executes the scripts, it seems the obvious place for values to be input as script parameters. However, no mapping should occur from DEVS to the scripts within the timer model. For example, the IM could send the output it receives from the agent model directly to the timer model and allow it run mapping functions such as f_{in} on this data. The problem is that this breaks the encapsulation of the timer with respect to its purpose — strictly timing control — and also decentralizes the interaction model, which increases the complexity of interaction management; neither of which is desired. The IM should provide the input that the CCA function scripts require. Given the large amount of data contained within each map, it would also be inefficient to send an entire map's worth of data as a message from the IM to the timer, in order for the timer to pass it to the input script. Consider, for example, that a 100×100 map has 100,000 cells and this number can easily go into the millions of cells. Sending 100,000 to over 1 million message objects is woefully inefficient in terms of computing resources. So too would be the creation of a data structure containing over 1 million data points and sending a reference that would have to be almost immediately unpacked and converted into a GRASS map. Instead, it is much more useful to employ the IM knowledge of the GRASS environment by allowing the IM to create a GRASS map that represents the output from the agent, mapped to $\overline{X}_{CCA}$ using the CCA's own mapping function f_{in}. Thus, $f_{in} \circ g_{in} : \Theta_{output} \mapsto \bigcup_{(i,j,k)\in D} \overline{X}_{ijk}$, discussed in section 3.1.2, all occurs within the IM. Similarly, $g_{out} \circ f_{out} : \bigcup_{(i,j,k)\in D} \overline{Y}_{ijk} \mapsto \Theta_{input}$ would

also be implemented within the IM. With this approach, the execution of the input mapping function script, f_{in}, and the output mapping function script, f_{out}, could be ignored. The IM will have already performed the mapping, ensuring that the interaction between the composed models remains encapsulated within the IM.

CHAPTER 4

FRAMEWORK IMPLEMENTATION

aeScape is comprised of four models (see figure 4.1). Two of the models, the Sugar CCA and the Landscape CCA, make up the environmental model. Both, as their names suggest, are composable cellular automata. To avoid confusion, "landscape" will be used to refer specifically to the Landscape CCA model, whereas "environment" will imply the entire aeScape environment model (*i.e.*, both the Sugar CCA model and the Landscape CCA model). Both CCA models are implemented in GRASS as a set of 100 cell × 100 cell raster maps and, as explained in section 3.2.2.5, use DEVSJAVA to provide timing for the CCA dynamics. DEVSJAVA is also used to model the agents and the interaction model. As figure 4.1 shows, the two CCA models interact directly with one another via their mapping function sets, F' and F''' (see section 3.1.1). Each CCA is also coupled to the agent model through the IM. F'' is defined within the Sugar CCA specification and F'''' is defined within the Landscape CCA specification. However, as shown in figure 4.1, these mapping functions can be implemented within the IM. This approach provides a more centralized interaction implementation, which provides more visibility into the interaction and supports interaction management.

Note that the mapping functions between each CCA and the IM is distinct for each CCA, and from the mappings between the CCAs. The mapping sets G' and G'' denote those that the IM employs, if necessary, to map I/O between the DEVS agent model and the mapping set F used by a CCA model. If the agent model is considered to be the external system Θ, then the G mapping function sets correspond to the mapping function set G discussed in section 3.1.2. Note that no CCA-to-CCA interaction occurs through the IM.

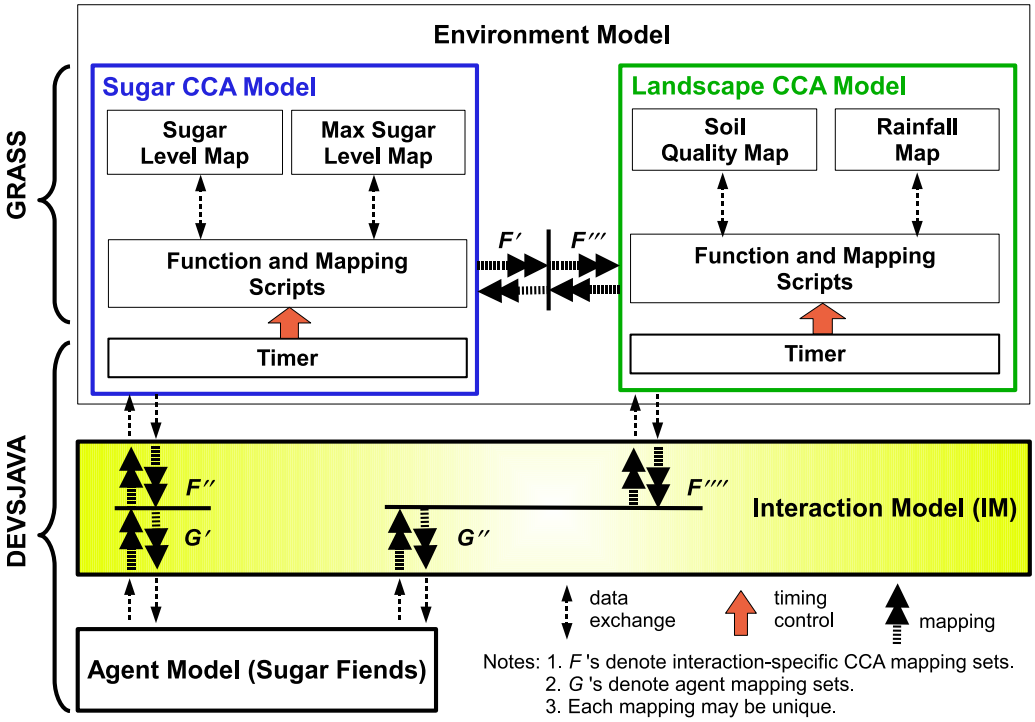

Fig. 4.1. aeScape — an exemplar agent-environment hybrid model.

The IM is only used for interactions between a specific CCA and the agent. This is because the two CCA use the same specification. Therefore, they have the same model structure, dynamics, and interface. An IM could be used between the two CCA to manage domain-specific disparities and/or maintain a clear separation of model concerns in order to enhance model reuse, as examples. For the purposes of highlighting the difference between models composed using an IM, and those that directly interface, the two CCA models will employ a direct interface. The interaction between these two CCA models implemented in this manner is mono-formalism, discussed in section 2.3.1.

As the agent model, the IM, and the two CCA are all using DEVSJAVA models for timing, a single DEVSJAVA simulation environment can be used to simulate the exemplar model. This reduces the need to synchronize multiple simulators. Assuming no timing

disparities between the agent and environment models, a general sequence of execution between these models is as follows:

- The CCA models generate output from their cells. Referring back to section 3.1.2, this output set is $\bigcup_{(i,j,k)\in D} \overline{Y}_{ijk}$, for each CCA.
- For the CCA-to-CCA interaction, the Sugar CCA applies $f_{out} \in F^{'}$ to its output set and the Landscape CCA applies $f_{out} \in F^{'''}$ to its respective output set. These mapped sets are then sent to the other CCA.
- For the CCA-to-agent interaction, the unmapped output sets from each CCA is sent to the IM. The agent model also sends it output data to the IM.
- The IM conducts the appropriate mappings in order to send CCA data to the agents and vice versa. For the agent-to-CCA data, this involves $f_{in} \in F^{''}$ and $f_{in} \in F^{''''}$. For the CCA-to-agent data, these mapping functions[1] are $g_{out} \in G^{'}$ and $g_{out} \in G^{''}$.
- The IM then sends the mapped output to the respective models.
- The composed models receive the external input and then update their state in accordance with their respective state transition functions.

More specific details about each of the model follows.

4.1 Environmental Model

The environment model has been made more complex than the basic Sugarscape model in order to demonstrate an interaction between two CCA models. Every location (cell) grows sugar crop. The sugar crop is dependent upon environmental factors to determine

[1]This maintains the CCA-centric "in" and "out" subscripts established in section 3.1.2

the amount of sugar growth every cycle[2]. Furthermore, the soil quality changes relative to how recently sugar was harvested at that location. Sugar growth, in turn, is dependent upon the soil quality.

The Sugar CCA is a model of the amount of sugar present and its growth. There is a maximum value defined for the sugar crop at each location. This maximum value is specified prior to the start of the simulation, and each sugar crop's maximum value remains constant throughout the simulation. Each cycle, if a sugar crop is below it maximum, it grows back by some base amount, scaled either up or down by the soil quality and rainfall in that location. Furthermore, to demonstrate the cellular automaton nature of the model, sugar crops propagate. Each cycle, a location sends out an amount equivalent to a fraction of its current sugar level to its nearest neighbors, the locations it influences. A location adds the average of the sugar propagations from neighboring locations, its influencers, to its sugar level growth. Cycles in which the sugar crop is harvested by a fiend, the sugar level is reduced to zero and does not grow.

The Landscape CCA models the environment in which the agent and sugar live. It contains values for soil quality and rainfall. Rainfall is arbitrary. It can be a constant or random and has no dependencies on other aeScape models' attributes. Soil quality, as mentioned above, is dependent upon the sugar levels at each location. When fiends harvest the sugar, the soil quality is reduced. As the land is left untouched by fiends, and the sugar begins to grow, soil quality begins to go back up. The soil quality is also dependent upon the average soil quality of its neighboring cells. Each cycle, after the external factors of sugar level and harvesting are accounted for, the soil quality settles toward the average by some maximum amount (if the difference is great, else it just becomes the average value).

[2]As in section 3.2.2.5, the term "cycle" describes the amount of simulation time that passes between a model's internal state transition.

4.1.1 Environment models' architecture

4.1.1.1 General CCA architecture for untimed models

As described in section 3.2.2.5, GRASS does not have an innate concept of time. To provide timing for the CCA exemplar models built in GRASS, a DEVSJAVA timing model is employed. Figure 4.2 shows a Unified Modeling Language (UML) class diagram of the general architecture that uses a DEVSJAVA timing model to provide time structure to the dynamics of an untimed model. First, though not shown in the figure, the AEScape class is a DEVS coupled model, and CCATimer is a DEVS atomic model. IComposableCA is an interface that defines specific method signatures that all CCA must expose in order for the timer model to provide proper dynamics to the CCA model. ConcreteCcaAdapter is an arbitrary class that implements the actual ("concrete") method functionality that are described by the interface method signatures. By using the interface in this way, the timing mechanism is explicitly separated from the model behavior. This allows flexibility in the modeler's choice of CCA software development environment.

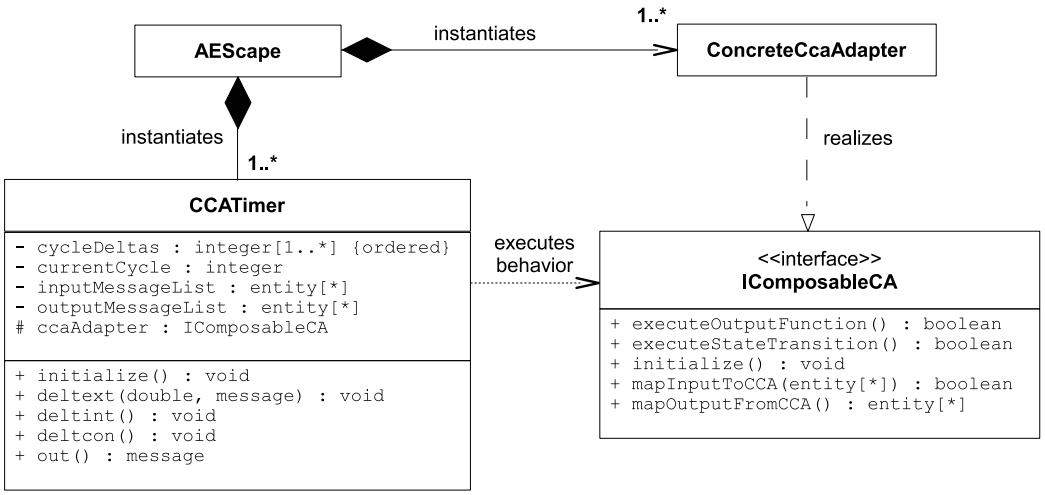

Fig. 4.2. aeScape general CCA class diagram.

CCATimer implements the standard DEVSJAVA atomic model methods of `deltext(), deltint(), deltcon(),` and `out()`. As discussed in section 3.2.2.5, the CCA-Timer has two states: input and output. The CCATimer's state transitions are specified such that it begins and remains in the output phase for the duration of the initial discrete-time segment delta, t_0. CCATimer then transitions to the input phase and stays there for zero-time before transitioning back to the output phase. To meet the CCA dynamics specification prescribed in table 3.1 and figure 3.4, CCATimer executes the IComposableCA methods of the concrete CCA adapter at the appropriate times within the standard DEVSJAVA model methods.

When CCATimer model is initialized to the output phase, it immediately undergoes a state transition, calling `deltint()`. This increments the model's internal cycle count and causes it to become passive if the last discrete-time segment delta has been reached. If the model does not become passive, it then schedules itself for an internal state transition to the input phase for zero-time. As part of the state transition from the output phase to the `input` phase, and immediately before the model enters the input phase, the CCATimer model's `out()` function is called. The `out()` method, when the model is in the output phase, calls the IComposableCA interface methods `executeOutputFunction()` and `mapOutputFromCCA()`, which are the functions λ and f_{out} described in section 3.1.1, respectively. The CCATimer then enter's the input phase. If an event (*i.e.*, external input message) is scheduled for the CCATimer at the same time that it schedules the internal transition to the input phase, then the CCATimer confluent transition function is called.

The CCATimer model's confluent transition function, `deltcon()`, must be set to specify that if an external message arrives at the same time an internal state transition is scheduled, the internal state transition will occur first, then the message will be handled.

This allows CCATimer to enter the input phase and update the current discrete-time segment. It can then properly manage the input data from the other models at the start of that discrete-time segment. If a message is received, CCATimer calls the `mapInputToCCA()` interface method within its `deltext()` method. `mapInputToCCA()` is the CCA implementation of the f_{in} function. The CCATimer then undergoes an internal state transition back to the output phase. As was the case from output to input, before the state transition occurs, the CCATimer's `out()` method is called by the simulator. Since the CCATimer is in the input phase, it calls the `mapOutputFromCCA()` interface method, δ from section 3.1.1.

In order for the two CCA to directly interact, they must have explicit knowledge about the domain representation that the other uses. For example, currently the Sugar CCA and Landscape CCA have the same spatial resolution. The Sugar CCA uses rainfall to scale the amount of sugar regrowth each cycle. It does this by taking rainfall output (in mm per cycle) and converting this map with a range of real values to a set of three values, each representing either high rainfall, medium rainfall, or low rainfall. If the Landscape CCA were to change the units in its output map and change the resolution at which the data was given, the Sugar CCA would have to be modified, potentially making significant changes to the way that rainfall is used in its calculations. Furthermore, error checking the interface between the currently requires following the map development path through the output steps of one model and then through the input steps of the other.

4.1.1.2 GRASS-implemented CCA architecture

Figure 4.3 shows a GRASS-implementation of the CCA architecture using the DEVSJAVA timer model. AEScape, CCATimer, and IComposableCA are the same as in figure 4.2. GrassCcaAdapter is a concrete implementation of the IComposableCA interface. Of key importance is that this adapter instantiates an instance of the GrassFacade class. This class

is *the* interface between the Java Runtime Environment (JRE) and the GRASS environment. The façade class wraps many of the commonly used GRASS modules within a Java method. Other Java classes call these façade methods (such as `rMapcalc()` and `rInfo()` for the GRASS `r.mapcalc` and `r.info` methods, respectively), passing input parameters as arguments, and the façade method will call the appropriate GRASS module without creating a script. The GRASS module output is then captured from the buffer stream and returned to the calling function as a Java String object for parsing. An important feature of the GrassFacade class is that it is a *singleton* (denoted by the "1" in the upper right-hand corner of its class diagram). Being a singleton means that only one instance of the object can ever be created at a time. This ensures that if multiple GRASS CCA are created at the same time (as they are in the aeScape exemplar model), they will all use the same GRASS environment (GRASS version, location, mapset, etc.). There are two subclasses of the GrassCcaAdapter, a SugarCcaAdapter and a LandscapeCcaAdapter (neither is shown in figure 4.3). These adapters define the specific initialization, state transition, output, and mapping functions for the Sugar CCA and Landscape CCA, respectively.

Figure 4.4 is a sequence diagram that shows the dynamical relationships between the CCATimer, GrassCcaAdapter, and GrassFacade classes. Note that if multiple GrassCcaAdapters exist, or if an interaction model is used, they would all request and receive the same instance of the GrassFacade. As explained previously, this ensures that all of them use the same GRASS environment and, specifically, mapset. This all occurs as the model objects are instantiated. Once the models exist, the DEVSJAVA simulator (not shown in figure 4.4) begins the process of coordinating events. This is expressed in the sequence diagram through the continuous processing that occurs within the CCATimer atomic model (the wide, vertical bar). As the CCATimer initializes (by a function call from the simula-

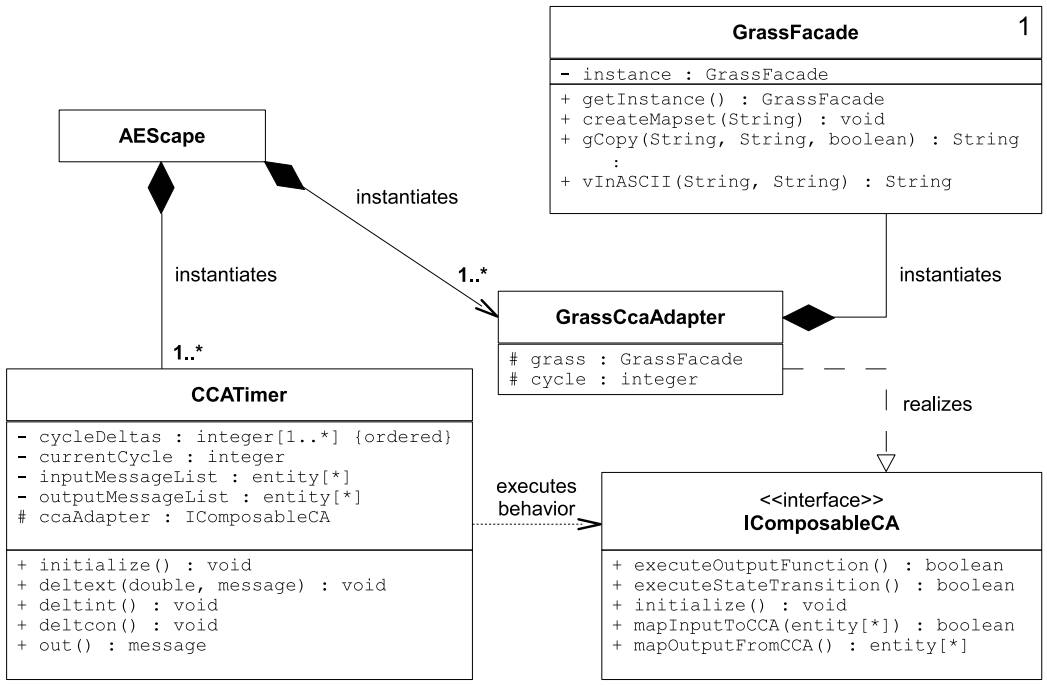

Fig. 4.3. aeScape GRASS-implemented CCA class diagram.

tor) it calls the `initialize()` function of the GrassCcaAdapter, which in turn creates the initial map (or maps, through multiple `rMapcalc()` calls) required by that specific CCA (*e.g.*, initial values). The remainder of the sequences are in a loop.

The label `loop |T|` in figure 4.4 specifies that the remaining sequences will be repeated $|T|$ times, where $|T|$ is the cardinality of the set T from equation (3.1) (*i.e.*, the number of discrete time segments for this CCA). This conforms to what is specified in algorithm 3.1. For each traversal of the loop, the CCATimer first calls the GrassCcaAdapter's `executeOutputFunction()` function to generate the output from each cell to its influencers, and to generate each cell's external output, if any — shown as `influenceMap` and `outputMap`, respectively. The term "`expr`" being shown passed as an argument to the façade's `rMapcalc()` method is the map algebra expression that the GRASS `r.mapcalc()` function uses to create the map. Parameter `expr` is provided by the specific GrassC-

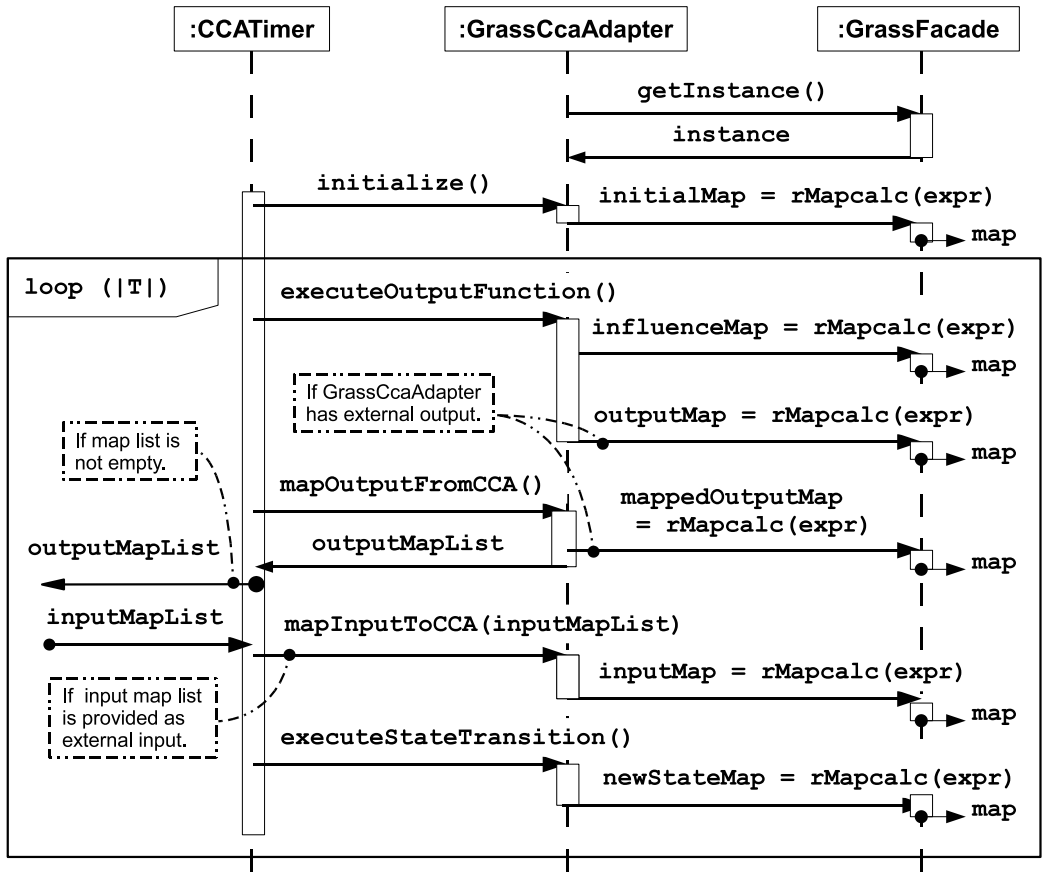

Fig. 4.4. aeScape GRASS-implemented CCA sequence diagram.

caAdapter implementation being used within the CCA. Next, the CCATimer calls the `mapOutputFromCCA()` function that maps the cells' external output to the network output, and returns a list of the names of the output maps that were generated. If the GrassCcaAdapter has no external output, the `mapOutputFromCCA()` function returns immediately and provides an empty return list. If at least one output map is created, the CCATimer sends out a DEVSJAVA message to the model to which it is coupled (*i.e.*, the IM or another CCA), notifying it of the external output maps that have been generated.

At this point in the sequence diagram, the CCATimer would have completed its internal transition to the input phase (see section 3.2.2.5). The CCATimer phases are not

shown in the sequence diagram for clarity. However, it is at this point that if the CCA-Timer receives an event from an external model (*i.e.*, the IM or another CCA), notifying it that external input is available, that it will then execute the `mapInputToCCA()` function, and pass the list of external input map names that it receives onto the CCA. Finally, the CCATimer calls the GrassCcaAdapter's `executeStateTransition()` method. The Grass-CcaAdapter generates a map algebra expression that takes into account the influence map, and the mapped external input, to generate a new map that represents the CCA's state. As discussed in chapter 3, the state of the CCA may be comprised of multiple variables, and each variable is represented within a separate GRASS map. Thus, the `rMapcalc()` method call would be repeated for each map that needed to be changed. Both the Sugar CCA and Landscape CCA follow this general sequence. Their specific behaviors are realized within the SugarCcaAdapter and LandscapeCcaAdapter. These two adapter objects will create different maps, generate different output, and account for different input through the use of distinct CCA logic and map algebra expressions. The specific attributes and behaviors of the Sugar CCA and the Landscape CCA follow.

The Sugar CCA model's state is defined by the sugar level at each of its locations. The sugar level value at all of the locations is stored in a GRASS raster map. When the Sugar CCA is initialized, it creates an initial sugar level map that represents its state at time t_0. The maximum value for the sugar level at each location is specified in a separate map. At the end of the discrete-time segment for t_0, the CCATimer for the Sugar CCA calls the Sugar CCA `executeOutputFunction()` method, which is the output function, λ, in the SugarCcaAdapter. This method creates $\dot{Y}$, each location's output to the locations that it influences, and stores it in a GRASS map. $\dot{Y}$ represents the sugar crop propagation and is equal to some fraction of the current sugar value. Next, the CCATimer calls the

Sugar CCA's f_{out} function, the `mapOutputFromCCA()` method. This method creates another GRASS map that represents $f_{out}(\dot{Y})$. In this case, the mapped output map has equivalent values to the $\dot{Y}$ map. The CCATimer then transitions to the input phase for zero-time. If as the CCATime is transitioning to the input phase it receives external input, the CCATimer will execute the f_{in} function method, `mapInputToCCA()`. The input messages are the names of the recently updated maps that represent output from external systems. Sugar CCA maps soil quality to a percentage value. Rainfall amounts are mapped to one of a set of three scalar values — 0.5 for a range of rainfall values considered low rainfall, 1.0 for medium rainfall, and 1.5 for high rainfall. Also, fiend harvest actions are mapped into a binary map of affected or not. Once in the input phase, the CCATimer schedules a transition to the output phase at a time equal to the duration of the next discrete-time segment. Before that transition occurs, the `out()` method is called, which executes δ, the `executeStateTransition()` method.

The Sugar CCA state transition calculates new values for the sugar level map at every location. The new value is dependent upon the action of the sugar fiends, environmental factors, and the health of the neighboring sugar crops. More specifically, let γ_n be the current sugar level at time t_n, γ_{n+1} be the sugar level at time t_{n+1}, γ_{max} be the maximum sugar level, $\psi = 1$ indicate that the fiends harvested the location and $\psi = 0$ that they did not, μ be the environmental increment, and ω be the neighboring sugar location propagation amount. Then,

$$\gamma_{n+1} = \begin{cases} 0 & \text{if } \psi = 1 \\ \min(\gamma_n + \mu + \omega,\, \gamma_{max}) & \text{if } \psi = 0 \end{cases}$$

where $\mu = (\text{regrowth constant} \times \text{soil quality } \% \times \text{rainfall scalar})$, and ω is the average of the propagation output of a location's influencers. In keeping with Sugarscape notation, this environmentally-dependent growback rule is called G_ε.

The state of the Landscape CCA model is defined by both the current soil quality value and a rainfall value at each location. Each value is stored in a distinct GRASS map using real numbers. Like the Sugar CCA, the Landscape CCA initializes by creating these two maps with their respective initial values at time t_0. The soil quality ranges between the values of 0 and 100, inclusive. The rainfall represents an arbitrary value of millimeters of rain fallen per cycle. The Landscape CCA output function generates a map representing output to influencees, $\dot{Y}$, that contains only the current soil quality. Two maps are generated for external output. They hold the values of the current soil quality and rainfall. When the external output maps are mapped from the cellular network using f_{out}, the rainfall map values are passed as-is. The soil quality map values, on the other hand, are rounded down to the nearest integer. The Landscape CCA only takes the sugar fiend harvest map as input. Like the Sugar CCA, the Landscape CCA f_{in} function maps these values to a binary representation. During the Landscape CCA state transition, the soil quality values are updated. The rainfall may be either constant, or random.

The new soil quality values in a Landscape CCA are dependent upon the sugar fiends' actions and the soil quality of surrounding locations. Let ρ_n be the current soil quality at time t_n, ρ_{n+1} be the soil quality at time t_{n+1}, $\psi = 1$ again indicate that the fiends harvested at that location and $\psi = 0$ that they did not, ν be the average soil quality of the location's influencers, c_h be some constant change due to harvesting, c_f is a constant representing unused land healing itself, and c_{max} is a constant maximum value the soil

quality change can undergo as it settles towards the neighborhood median. Then,

$$\rho_{n+1} = \begin{cases} \max(0.0,\, \rho_n - c_h) & \text{if } \psi = 1 \\ \min(\max(\rho_n - c_{max},\, \nu) + c_f,\, 100.0) & \text{if } \psi = 0 \text{ and } \rho_n > \nu \\ \min(\min(\rho_n + c_{max},\, \nu) + c_f,\, 100.0) & \text{if } \psi = 0 \text{ and } \rho_n < \nu \\ \min(\rho_n + c_f,\, 100.0) & \text{if } \psi = 0 \text{ and } \rho_n = \nu \end{cases}$$

The identifier V_β will be used to describe this landscape environmental rule where the impacts are not directly apparent to the agent, and to highlight the segregation between the landscape and the sugar growth. Thus, the environmental system is described by the set of rules $\{ G_\varepsilon, V_\beta \}$.

4.1.2 CCA interactions

As aeScape is a hybrid model, the two composed components of the model — the environment model and the agent model — can be executed independently. This provides the opportunity to examine the interaction between the two CCA independent of the other models. Figure 4.5 shows the interface between the two CCA where their interaction occurs. As depicted by the figure, data is not passed directly through the interfaces between the CCA models. GRASS maps are used as data exchange mechanisms. The message passing in DEVSJAVA is used to allow the CCA to communicate and inform each other when specific maps have been generated. Using the maps in this way takes advantage of the GRASS environment efficiency with large data sets that the maps represent. It also avoids having to convert all of the map data into an object, pass it to the other CCA, just to convert back to a GRASS map. However, in taking this approach, the modeler must ensure that one CCA does not directly modify the maps representing the state of the other CCA. Thus, the message passing between the models is used to inform the other model

when output data is made available. Combined with the fact that each CCA has its own distinct timer, this allows the two CCA to have different discrete-time segment deltas.

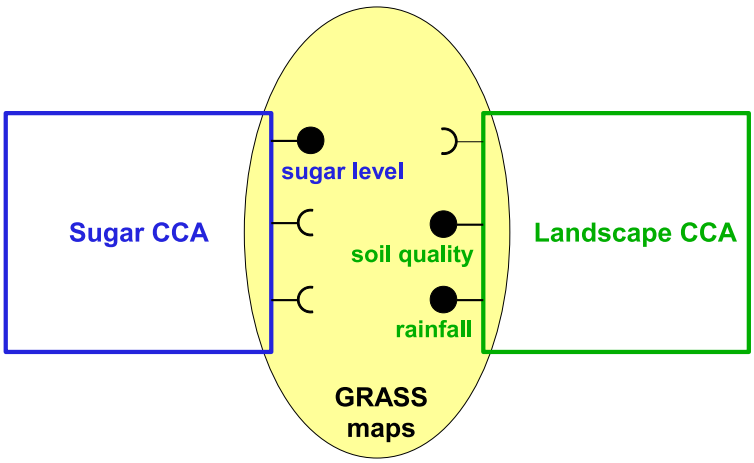

Fig. 4.5. Interaction between environment model CCA.

To prevent the CCA from directly modifying each other's state-representative maps, each step of the CCA dynamic process described in algorithm 3.1 is executed explicitly. In terms of GRASS mechanics, what happens is that the CCA creates a map of external output from each location as a result of the λ output function. This map can be referred to as $\overline{Y}$. Then, when the CCA's f_{out} mapping function is executed, the values in $\overline{Y}$ are mapped to new map representing the output from that CCA. Call this map Y_N. The second CCA then applies its input mapping function, f_{in}, to Y_N, treating it as its X_N and creating external input for its individual locations. As an example, the Landscape CCA stores the soil quality values as real numbers, which range in value from 0.0 - 100.0, inclusive. When it generates the external output map, $\overline{Y}$, it simply copies the values from the current soil quality map to $\overline{Y}$. However, when f_{out} maps these values to Y_N, only the integer portion of the value is mapped. It is this integer-valued soil quality map that the Sugar CCA uses as external input, X_N, and then applies its input mapping function, f_{in}, which divides the integer value by 100.0 to generate a percentage value that it uses to scale the growth rate

of the sugar. In this manner, the mapped output map becomes the interface for the two CCA, and it is assured that neither will directly manipulate the state-representative data maps of the other. This, in turn, maintains the validity of the models' behaviors.

4.2 Agent Model

Sugar fiends must consume sugar every cycle in order to survive. Each cycle the fiend looks around, moves to the closest location with the best sugar crop, and harvests the entire crop at that location. It then consumes some or all of the sugar it has in hand. If there is sugar left over, it can hold it in reserve. If two or more fiends move to the same location to harvest crop, there is one of two alternatives that the current model offers. The first alternative is combat from which, at most, only one fiend is declared victorious. The second alternative is that all of the fiends equally share the crop. One type of behavior must be specified at the start of the simulation, and that behavior will be exhibited by all fiends throughout the simulation. The combat behavior is explained in more detail in this section, below.

Every fiend has the same attributes that represent the state of the fiend model. A unique identification distinguishes one fiend from another. The location attribute is the fiend's current location. A fiend's vision range specifies how far a fiend can see, and consequently move, each cycle. The vision range is a constant value whose range is from 1 to 6, inclusive. A metabolism attribute specifies how much sugar the fiend must consume each cycle in order to survive. If the fiend does not have enough sugar to meet its metabolic requirement, it dies that cycle. The metabolism value is also a constant value, but whose range is 1 - 4, inclusive. A fiend's last attribute is sugar wealth. It is the amount of sugar that the fiend carries with it. The value is used to feed the fiend, and also determines the fiend's strength during combat. While the sugar wealth can not fall below zero, it has no upper bound.

All fiends have the same set of behaviors. Each cycle, a fiend examines locations as far as its vision range will allow it to see. This examination is done only in locations directly north, east, south, and west of the fiend's location. As simply a choice of abstraction, diagonal viewing is not permitted in this exemplar model. The fiends then move to the closest viewed location with the highest amount of sugar crop. If there are multiple, close locations with the same sugar value, one is chosen at random. Once at the new location, if there are no other fiends, the sugar crop is harvested. Once the sugar is harvested, the amount of sugar crop is added to a fiend's sugar wealth. Finally, the fiend eats an amount of sugar from its sugar wealth equal to its metabolism. As stated before, if there isn't enough sugar in a fiend's supply, the fiend dies that cycle.

If the combat behavior is enabled and more than one fiend moves to the same location, combat ensues. Combat is conducted by recursively pairing off the two most powerful, alive fiends at that location. The sugar wealth of the two fiends are compared. The fiend with the highest current sugar wealth wins, but has its current sugar wealth reduced by its opponent's wealth. The loser's wealth is reduced to zero and it is considered dead. If two fiends with the same sugar wealth combat, both die. If there are two or more alive fiends remaining after combat, then the next two most powerful fiends combat. If only one fiend remains alive, it harvests the sugar and adds the crop value to its remaining wealth. If no fiends remain alive, then the sugar crop is declared the winner as it is not harvested that cycle. As an example, four fiends go to the same location. Fiend-1 has a sugar wealth of 3, fiend-2 has 8, fiend-3 has 5, and fiend-4 has 2. According to the rules, fiends 2 and 3 combat first. Fiend-3 would die since it is weaker than fiend-2, and fiend-2's sugar wealth would reduce to 3. The next two most powerful fiends are fiend-1 and (again) fiend-2, both with

a sugar wealth of 3. Since their wealth is equally powerful, both die. This leaves fiend-4 as the victor, and capable of harvesting the sugar crop uncontested.

Sugarscape uses "M" to describe a basic agent movement rule. The fiends employ this rule, but with the added behavior of combat or sharing. Therefore, movement rule M will be expressed here as a set of possible movement rules, $M = M_c, M_s$, where M_c is the movement rule that results in combat when two or more fiends occupy the same location, and M_s is the movement rule that results in sharing when two or more fiends occupy the same location.

The agent model (shown in figure 4.6) is comprised of a single DEVSJAVA atomic model, a collection of Fiend objects, and a Location object for each Fiend. In the exemplar model, the fiends use the same resolution and location designation as the environment models. Thus, the xCoordinate and yCoordinate Location class attributes contain values that correlate directly to a location in the environment maps. As every fiend has the same attributes, exhibits the same behaviors, and has the same same behavior timing; a single DEVS atomic model is used to manipulate the Fiend objects that represent an individual fiend. The eat and sugar wealth comparison behaviors are provided within the Fiend class itself. The location examination, movement, and harvest behaviors are defined within the IM due to the dependency of both agent and Sugar CCA model data. Each cycle, the SugarFiends atomic model sends the IM current attribute data for each fiend in order to conduct the landscape examination, movement, and harvest behaviors. The IM then sends the updated information back to the SugarFiends atomic model and it updates each Fiend object. The `eat()` method of each fiend is then executed and fiends with sugar wealth less than zero are removed from the collection of fiends.

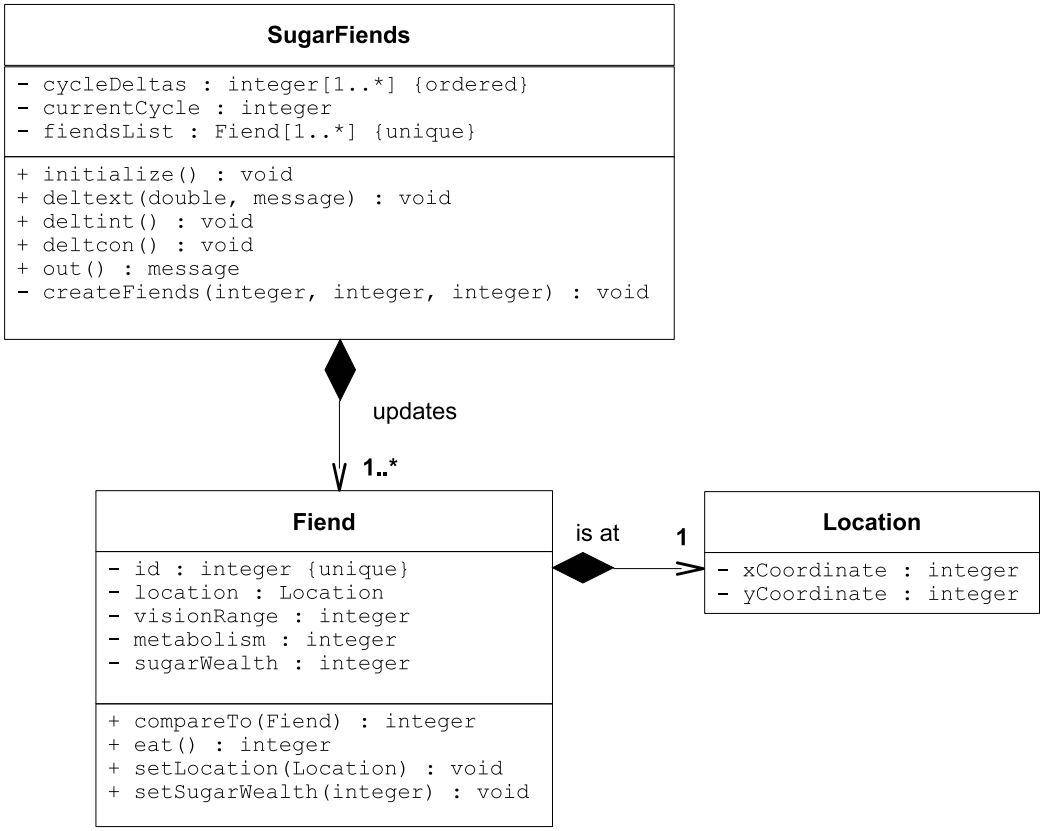

Fig. 4.6. Class diagram of aeScape agent model.

4.3 Interaction Model

The IM is a single DEVS atomic model. It models the interaction between the agent model and the Sugar CCA, and between the agent model and the Landscape CCA (see figure 4.1). The IM's state set, $S = \{$"passive", "send"$\}$. The passive state represents the IM's initial state and the state in which it remains when mapping is complete and it is awaiting further input. When the IM receives input, the external transition function maps the input to the associated output in zero simulation time. It then undergoes a state transition to the send state in order to send the output immediately. If no further input is received, an internal transition returns the IM to the passive state. The IM is given three input ports ("agent_in", "lscape_in", and "sugar_in"), and three output ports ("agent_out",

"lscape_out", and "sugar_out"), for the input/output from/to the agent model, the landscape model, and the sugar model, respectively.

The aeScape domain can be applied to the equations described in section 3.2.3, in order to provide more specific details about the IM. $X_{IM} = \{\Phi, \Gamma\}$, where Φ is a collection of fiends and their associated attributes (location, vision range, sugar wealth, and metabolism), and Γ is the set of values representing the current sugar levels at each location. $Y_{IM} = \{\phi_h, \phi_\ell, \phi_w\}$, where ϕ_h is where the fiends harvested sugar, ϕ_ℓ is the fiends' new locations, and ϕ_w is the fiends' new sugar wealth. The IM contains two mapping functions, both from the agent to the environment model. Thus, $W = \{w_1, w_2\}$, where $w_1 \equiv \Phi \mapsto \phi_h$ and $w_2 \equiv \{\Phi, \Gamma\} \mapsto \{\phi_\ell, \phi_w\}$. In addition to expanding these mappings to include a specific input port-related function (as discussed in section 3.2.3), the mappings can further be defined by the output port.

As with the input ports, specifying the output port to which the mapping applies improves the modeler's capability to manage the mappings within the IM. Similar to the input to the mapping function, the output can be expressed as a port-output pair. More specifically, $w_1 = f(\{\text{"agent\_in"}, \Phi\}, \text{"passive"}, t) \mapsto \{\{\text{"lscape\_out"}, \phi_h\}, \{\text{"sugar\_out"}, \phi_h\}\}$, and $w_2 = f(\{\{\text{"agent\_in"}, \Phi\}, \{\text{"sugar\_in"}, \Gamma\}\}, \text{"passive"}, t) \mapsto \{\text{"agent\_out"}, \{\phi_\ell, \phi_w\}\}$. As before, $ta() \vdash t$. Note that including the output port provides visibility to the fact that the output from w_1 goes to both the Landscape CCA and the Sugar CCA.

Each cycle, when the agent model sends the current fiend data, the IM iterates through the collection of fiend data. For each fiend, it queries the Sugar CCA locations that are within the fiend's vision range from its current location. It then determines the closest location with the highest current sugar crop. Fiend combat is resolved and sugar crop is harvested, as appropriate. New location values and sugar wealth values are provided

back to the agent model in order to update the state of each fiend. In addition to assisting the fiends in their movement and harvesting, the IM provides input for the Sugar CCA and Landscape CCA based upon the fiends' actions. Once the new fiend locations are determined, the IM creates a map specifying which fiends harvested at which locations. Using GRASS maps to store agent data is a convenience enabled by the IM having data for both models and the knowledge of both model's domains. From a modeler's perspective, the agent location data is easier to understand as there is a direct correlation between map value and the spatial representation of the sugar and landscape locations. Furthermore, devising a map of agent harvest activity makes it easier to provide agent activity data as input to the two CCA.

From a CCA perspective, w_1 is the g_{out} that was discussed in general terms in section 3.1.2. The fiend harvest output map, ϕ_h, is the result of that function. As the IM is the model of the interaction between the agent model and each CCA, the IM contains the input mapping functions for agent harvest activities for each CCA. Thus, the IM applies the Sugar CCA's f_{in} mapping function to ϕ_h to create $\overline{Y}_{sugar}$, and also applies f_{in} from the Landscape CCA to ϕ_h to create a second map, $\overline{Y}_{landscape}$. The Sugar CCA and Landscape CCA then use their respective maps as external input during their next state transition.

In the best case, where the whole interaction can be encapsulated within the IM, the IM itself will have mapped the external output from the CCA network and the external input to the network. Thus, referring back to figure 4.4, the `mapOutputFromCCA()` and `mapInputToCCA()` functions will do nothing with respect to mapping data from and to the agent model, and return immediately. Note that they would still have to map I/O for direct CCA-to-CCA interactions. If the model abstraction does not allow the IM to encapsulate

the entire interaction, or a CCA is being used within a modeled system that does not employ an IM, then all of the I/O mappings would occur within these functions.

CHAPTER 5

APPLICATION

The hybrid agent-environment model, composed using poly-formalism composition, has been applied in two scenarios. The first is in the exemplar model, described in chapter 4. The second application is in the MedLand project. Both applications are discussed to some detail in this chapter.

5.1 aeScape Application

The intent of the aeScape model is to provide some proof of the concepts presented in this dissertation, in a domain to which some may have some exposure (Sugarscape) and, therefore, familiarity. The ability to develop, compose, and manage the hybrid aeScape model is what was being tested. To this end, the models were built according to the proposed hybrid agent-environment model, and simulations were successfully run. Figure 5.1 shows the hybrid agent-environment model in CoSMoS.

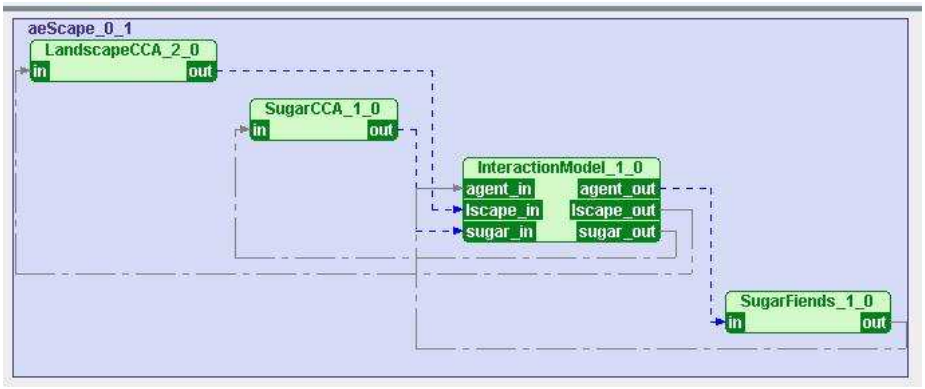

Fig. 5.1. aeScape hybrid model shown in CoSMoS.

Multiple simulations of aeScape were run, each with 50 cycles (discrete segments). All three models were provided the same, constant discrete-times. For the CCA, this means that all 50 discrete-time deltas were given as the same value. For the agent, the internal state transitions were devised such that the agent repeated the same actions in a manner that coincided with the CCA discrete-segments. Simulations were run using different agent

behaviors, varying initial fiend populations, different initial sugar values, different initial landscape soil qualities, and different sugar value maximums with different patterns. The maximum value "patterns" included using the same value for all locations; setting a percentage of all locations (randomly chosen from the 10,000 cells) with a high value, while setting the remainder to a much lower value; using a varying width equator line running across the landscape on which sugar "grew better"; and having a large central map location with high maximum sugar values and a much lower value at all other locations.

The explicit separation of sub-system models eased the development process. The environment CCA were built first, and independently. Using constant-valued maps to represent anticipated input values, the individual dynamics of each CCA could be developed and tested for consistency and correctness of behavior. Next, the two CCA were composed directly with one another (as shown in figure 4.5). At first, a constant-valued input map was used to represent fiend harvest actions. Then, the IM was added to emulate external input. The fiend model was devised alongside the IM. The IM and fiend model were also tested together with the IM emulating the environment model. Once the agent and environment models appeared to be operating correctly, they were composed using the IM and all test maps were removed. While the models simulated correctly, once all of the sub-systems were interacting, it became clear that some of the abstractions made would require revision.

Modifying the behaviors of the models was made simpler because of the KIB approach. For example, the environment model was recovering more quickly than desired from fiend harvest actions. It took little effort to revise the Sugar CCA and Landscape CCA, retest the model behaviors from an individual model level to sub-system models composed as part of the hybrid model. Furthermore, all of this was done without having to revise the fiend agent model. As another example, the fiend model originally only had combat behav-

ior. The share behavior was easily added by modifying the agent-environment interaction defined in the IM. Neither the environment nor the fiend model required code modification. This is because all of the data required, where the fiends were located, how much sugar was available for harvest, etc. was already centrally located within the interaction model. Thus, variations aeScape of rule-sets $(\{G_\varepsilon, V_\beta\}, \{M\})$ could be accomplished with minimal impact to the models within the hybrid system model.

Simulations of disparate discrete-time segments were also run. The Sugar CCA and agent model were provided a synchronized timing scheme, as before. The Landscape CCA's discrete-time segments were scheduled to occur half as frequently as the Sugar CCA and agent. To manage this disparity, the Sugar CCA had to be revised to make use of the old Landscape CCA soil quality and rainfall data. In the exemplar model, the Landscape CCA did not rely upon the Sugar CCA for input. If it had, then the Landscape CCA model would have operated on a granularity of half of what was being produced as the Landscape CCA would have ignored all input from the Sugar CCA received during the middle of one of its discrete-time segments. Alternatively, the Sugar CCA would have had to compensate for the timing disparity by maintaining a unique input map that could be sent to the Landscape CCA during the appropriate times. Obviously, this approach would require that the Sugar CCA have intimate knowledge of the Landscape CCA behavior, which implies less freedom in the Sugar CCA abstraction and modelers implementation. Furthermore, if the timing of either model were again changed (for example, if the Sugar CCA were then set to occur only every third cycle), the two models would have to be re-examined and possible redesigned to support the timing changes. This is because the Landscape CCA model would now have the faster state change cycle, and it might now be more appropriate to move all of

the additional synchronization logic into this model. This is much more cumbersome than what must occur for the Landscape CCA-to-DEVS agent interaction.

The interaction between the landscape and the agent occurs within the IM. Given the original timing disparity example in which the agent state changes occur twice as fast, the IM could compensate for this by maintaining the two sets of agent actions that occur for each Landscape CCA discrete-time segment. Neither the agent model nor the landscape model would require modification. This would allow these models to continue to be abstracted in a manner that best represents their specific system within the domain. While this seems similar to what the Sugar CCA had to do, there are benefits to the use of the IM. As the IM also handles the interaction between the sugar model and agent model, the IM could use its knowledge of the landscape timing disparities to compensate for the lack of input to the sugar model. For example, since the agent's actions would reduce the health of the landscape, the IM could apply some penalty to account for what the agent did in the previous cycle that is not yet reflected in the actual landscape model. Furthermore, if the timing disparity changes, like the Sugar CCA using one-third of the agent's cycle, only the IM would change and any dependencies that the disparate timing mechanisms of the landscape and sugar models would have on the agent model could all be accounted for within the IM. The agent model would not grow any more complex. This maintains a clear visibility into the interaction management and affords better reuse of the composed models.

Collection of simulation data was also made simpler by the use of an interaction model. A data collection method was added to the IM to capture fiend location changes and details about behavioral outcomes, such as which fiends died as a result of combat versus lack of sugar. The use of modeling formalisms and a formal approach to composing the sub-system models provided a very structured environment from which to gather data. Data

and structures within the hybrid model are not arbitrary. It's purpose within the hybrid model is clearly defined and, therefore, it makes it easier to understand the applicability of the data to the information that is desired from the model. The interaction model, due to its knowledge of the composed models via mapping set W (see section 4.3), is the optimal place to acquire data about the interaction itself, without having to explicitly design any of the models around the data collection process.

In summary, the aeScape exemplar model proved that the specification and algorithms for a CCA presented in chapter 3 could be implemented in a rigorous manner, within the GRASS development environment. Furthermore, these CCA could then be formally composed with a discrete-event formalism to devise a hybrid model. Additionally, the benefits of using a poly-formalism architecture, in which the sub-system models are composed via an interface, were proven beneficial to the development, testing, and revision of the sub-system models and the hybrid model as whole. Finally, by explicitly modeling the interaction between the composed agent-CCA models within a single model, a convenient location is provided for collecting data about the interaction that does not require specific sub-system model revision.

5.2 MedLand Application

The poly-formalism composition approach to composing an agent and environment model fits well with the needs of the MedLand project. MedLand requires an approach in which the agent and environment models can be developed by different groups with different expertise. The hybrid model is to be used as a laboratory setting in which to test hypothesis about data and system dynamics. As such, all models within the hybrid model are subject to revision. Therefore, model development and testing is improved if all of the models are to maintain some independence from the others, as this reduces the propagation of im-

pacts caused by making changes to a specific model. Furthermore, the dynamics within the sub-system models are complex. This complexity is compounded when the sub-system models are composed. An interaction model would help manage the complexity generated by the interaction between the agent and environment models. While MedLand is a multi-national, multidisciplinary project; research efforts developing both the agent models and interaction model for the project provided a lot of the knowledge and experiences required to conceptualize and devise the agent-environment composition. Additionally, as previously stated, MedLand provides a real-world example of simulation environments in which poly-formalism composition can be very helpful. Therefore, it is useful to describe the project to some detail, as it should aid in the understanding of some of the complexities that an interaction model can help manage.

The MedLand human-environment hybrid model is a simulation of human farmers living in the Penaguila Valley, Spain, during the early Bronze Age (Soto et al. 2007; Ullah and Barton 2007; Mayer et al. 2006; Ullah et al. 2008). The agent model represents human farmers who employ cropping cycles in a rain-fed subsistence agro-pastoral system. The landscape model is a process-based, cellular automata simulation of surface runoff flow, sediment detachment (erosion), sediment transport, and sediment deposition. The agent model is dependent upon some characteristics of the landscape model to make decisions and provide for its survival. The actions that an agent takes in order to survive may then impact the landscape model. These impacts change the dynamics of the landscape model and, quite frequently, the same characteristics upon which the agent depends. A clearer delineation between complexity emerging from within a sub-system model (*i.e.*, algorithmic and deterministic) and that which results from interaction with the other composed model

(*i.e.*, aggregate) is gained by directly expressing these models' interaction within a separate model.

With respect to a sub-system model, separating these *internal* and *external complexities* (see chapter 1) has other benefits. The algorithmic and deterministic complexities are more the result of modeler abstractions and implementation choices. These choices are dependent upon the experimentation setup and the domain. Thus, one may argue that there is some degree of modeler control over these complexities. On the other hand, aggregate complexity typically materializes as emergent behavior. As this behavior is not explicitly defined by the modeler and its results are often non-deterministic, there is much less modeler control. This becomes even more true when one considers that different modelers are developing the sub-systems models, and must coordinate their composition. Further, as the emergent behavior must be analyzed at the hybrid system level, there needs to be visibility and flexibility in the hybrid model design in order to test and modify the sub-system models such that the resultant emergent behavior, assuming that it is desired, produces valid results.

To understand the benefits of this approach to MedLand, it may be helpful to note that the environment model was developed in parallel with the agent and the interaction models, by different modelers. A number of domain experts provided models of systems from their respective fields (*e.g.*, erosion Ullah and Barton [2007], agent Mayer et al. [2006], and climatology Miller et al. [2008]). During the course of developing the environment and agent models, both used proxies for the other in order to test specific behaviors. The environment model was developed in GRASS and, as such, did not have the ability to easily create a very rich agent behavior set. The agents were relocated to new locations, within a specified range from a village site, in a stochastic fashion. The agents had no concept

of managing land and, therefore, no need for goal-oriented behaviors. While suitable for testing the landscape behavior, this type of agent behavior does not provide the capability to conduct studies from the agent perspective. Similarly, the agent model was developed in DEVSJAVA. An environment model was also developed in order to test it. However, the environment was very limited in scale due to the resource requirements of a large-scale environment developed in DEVSJAVA. Therefore, it would not have been much use to researchers who wanted to study large-scale environmental impacts. As the two models reached a stage of maturity, an interaction model was devised that enabled the environment and agent models to interact, within the frameworks that best suited their abstractions of the application domain. Composed together, researchers can ask questions about both models, and have a central place from which to manage the models' interaction without the requirement that any one model have dependency upon the other outside of domain applicability.

A rich user interface based upon the hybrid system model composition was added to the DEVSJAVA simulation environment to allow researchers to initialize model variables. The GUI provided flexibility in specifying domain/model knowledge throughout the hybrid model without having to resort to independently modifying multiple models before each simulation run. Traditionally, model initialization requires code development. The GUI makes the simulation environment more accessible, particularly for non-programmers.

Figure 5.2 shows the MedLand models in the DEVSJAVA SimView interface. The six households are the agents. The households that live within the same village are shown as such. Each village is a DEVS coupled model. As coupled models can not have attributes, such as total village population, an atomic model called a village manager (`VManager` in the figure) was created specifically for this purpose. It provides the village attributes and

behaviors. The `VgMsgPasser` and `HhMsgPasser` atomic models help manage message traffic from the interaction model (`Interactions` in the figure) to the households. The `Season` atomic model provides an environmental state that allows the agents to make decisions on what seasonal actions to take (*e.g.*, farm in the spring versus harvest in the fall). The interaction model serves a dual role. It does provide all of the mappings between the GRASS landscape models and the household agents. It also directly encapsulates the behaviors necessary to provide timing and input to the GRASS landscape models. Thus, the IM is not so well abstracted in this model as it was in the aeScape model. However, this has provided some flexibility in managing the timing of the GRASS landscape models, which do not follow an explicit modeling formalism nor have a specific interface beyond the individual GRASS modules.

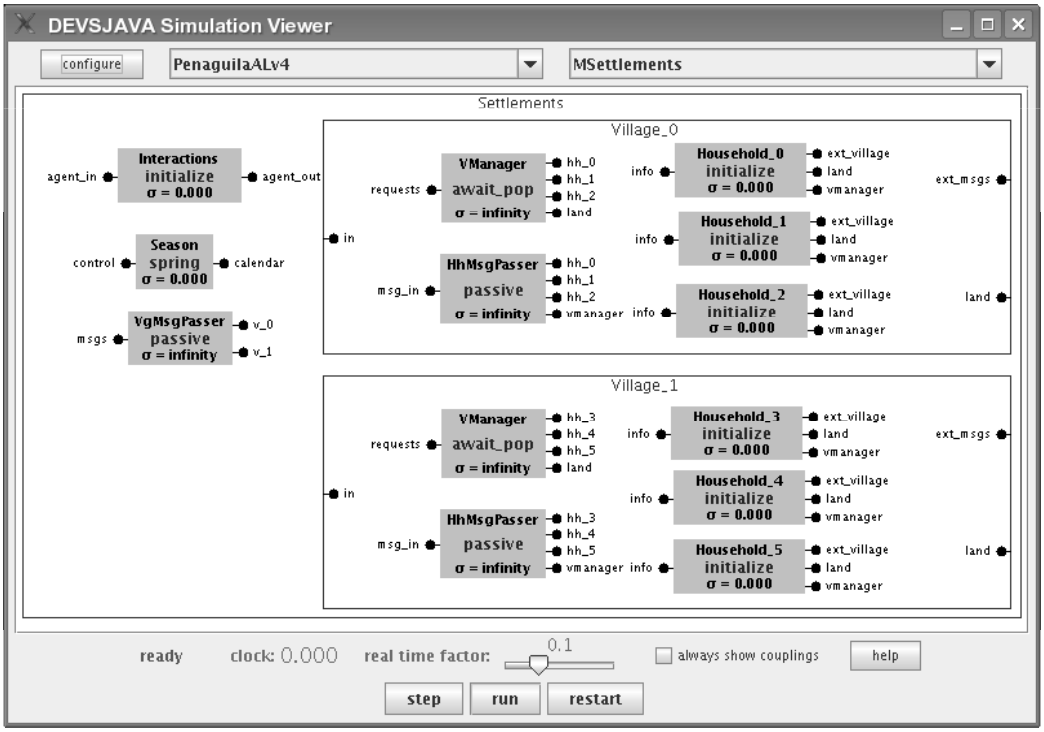

Fig. 5.2. MedLand hybrid model shown in DEVSJAVA SimView.

5.2.1 Agent model

The agent model is a combination of discrete-event, rules-based agents and models that define an agent's relation to another (Mayer et al. 2006). The discrete-event timing nature of the agents signifies that the agent conducts its actions in a discrete fashion but its current action may be interrupted by external factors. The result of such an interruption is dependent upon the agent's current state at the time.

The agent is a representation of a human household in which all members have common goals and share resources managed by the household. The model maintains population and allows for growth (positive and negative). The goal of each household is simply survival. A household is given a caloric requirement based on a per capita value. Additionally, the household is able to provide labor based upon all or a percentage of its population. To support its population, a household must feed its members by farming wheat and barley. The wheat is consumed directly by the household while the barley is used to feed livestock that produce milk and meat, which add to a household's caloric intake. Each cycle, the household calculates how much of each crop is required and how much land is required to produce it. Assuming that the household has enough labor to cultivate for the projected crop, it attempts to do so. If not, it attempts to cultivate as much as its current laboring populace will allow. To farm, households conduct a survey of their surrounding landscape. They assign a value to land based upon the attributes of soil depth, land cover, and distance. Once the land is assigned a value, a household makes a plan for that cycle which meets its needs (or comes close as possible) using the most valuable land. A probabilistic birth rate and death rate, modified by the ratio of yield to need, is used to determine a household's yearly change in population (if any) (Cowgill 1975; Wood et al. 1998).

Land cover is a discrete-time abstraction of both the plant growth on the soil surface and the health of the soil overall. Healthier soil supports more plant growth. The households desire a moderate level of land cover which is indicative of soil that is healthy enough to support a large crop without having to put too much effort into clearing the land of trees and shrub to cultivate it. If the land cover is above the desired amount, the households will reduce it to a desired level for farming, indicative of clearing the land.

5.2.2 Landscape model

The landscape model is based upon well known and broadly applied algorithms that calculate sediment flux at points on a landscape, given the supply of sediment at that point and the topographic characteristics of the areas up and downslope from that point (Braun et al. 2001; Dietrich et al. 2003; Hancock 2004; Mitasova and Mitas 1993; Willgoose 2005). These flux calculations also rely on the results of flow accumulation. The model simulates discrete flow of water and its accumulation in each cell. The sediment flux at each point on the landscape is then transformed into an elevation change through the Unit Stream Powered Erosion Deposition (USPED) equation (Mitasova and Mitas 1993), a three-dimensional modification of the Revised Universal Soil Loss (RUSLE) equations (Warren et al. 2005). In the landscape model, net erosion/deposition is derived from modeling the transport capacity of flowing water in the direction of greatest slope (topographical aspect). The change in slope and in upslope contributing area (catchment) across a given cell is used is to approximate the amount of erosion or deposition of sediment in that cell. A negative change (convexity) indicates increased transport capacity, and therefore erosion, while a positive change (concavity) indicates decreased transport capacity, and therefore deposition. The sediment transport capacities are modified by a transport factor, Tr, derived from a modification of the standard RUSLE equation (Ullah, I. A., pers. comm.). Tr is the

product of the rainfall intensity and three erosion resistance factors. These erosion resistance factors are a soil erosion resistance factor based upon soil composition, a vegetation erosion protection factor based upon vegetation's ability to hinder rainfall and surface flow, and an erosion prevention practices factor (*e.g.*, terraces and check dams). The rainfall intensity factor is computed by an equation that combines monthly precipitation amounts and is expressed as a map of rainfall intensity for each area in the landscape. The value is calculated from retrodicted precipitation values for the middle of Early Neolithic (7,000 BP (Before Present)) for southern Spain (Bryson and McEnaney-DeWall 2007). All of the erosion resistance factors are scaled from 0 to 1, with 0 being non-erodible and 1 being unprotected from rainfall intensity (Ullah and Barton 2007).

The landscape model also includes a simple vegetation model. Vegetation is defined on a 50-year time-scale. Any degraded land cover, if left unmodified by the agents, is regenerated in step-wise fashion through a simplified succession sequence. Bare ground will grow back through grass to scrub and eventually to forest cover in a 50 year period if no human impact occurs during the regeneration cycle (Pardo and Gil 2005). The land cover values are then translated to vegetation erosion protection factor values based upon linear regression of the known relationship between "classic" land cover types and vegetation erosion protection factor values in Mediterranean environments.

It needs to be stated that the current landscape model does not conform to the CCA specification. The CCA specification was developed in parallel with the landscape models and, as such, was not available when the landscape models were devised. Also, the current landscape models employ a neighborhood that is much larger than immediate neighbor for its calculations. A neighborhood with a radius larger than 1 has not been studied or shown to be correct within the CCA specification. Additionally, there is no concept of individual

cell encapsulation as is defined by the CCA specification. Therefore, cell state values can be manipulated directly by other models without a formal approach to state transition within the effected model. It's not that this can not occur in a CCA model such as aeScape, for example. It's just that this mindset was not employed as the models were designed and built. Therefore, while an interaction model can be used to compose the human and the landscape sub-system models, a guarantee of correctness can not be claimed from having followed mathematical specifications for each model and, therefore, the hybrid model as a whole. Additional management on the modeler's behalf is required in order to ensure the model behavior can be verified and the simulation output can be validated. The additional visibility gained by the use of an interaction model still assists in this endeavor.

5.2.3 Interaction model

Household farming impacts the environment by reducing the land cover. When coupled with the landscape evolution model, this has the effect of increasing soil erosion in that area. As soil erodes, less crop yield is produced by the same land. However, each year that land is allowed to fallow, the land cover increases. This represents some health returning to the soil. As plant matter grows back, it also has the effect of decreasing soil erosion, thus possibly increasing soil depth. An emergent result of this logic is that the households may exhibit the behavior of cycling their planting sites — farming in one area the first cycle and, then, farming in another area in a later cycle and allowing the previous site to fallow. Additionally, the village footprint also impacts erosion. While no crops are grown within the village itself, villages were often established near water sources. The erosion change caused by the village often has effects on the environment downstream.

The IM encompasses all of the interaction occurring between the agent model and the landscape model. In the current hybrid model, there are four interactions each simula-

tion cycle: the agent view of the landscape, the agent impacts on the landscape, the agent's harvesting of the crops, and the landscape processes affected by agent actions. It is these four interactions which define the aggregate complexity for this hybrid model. For each of these, data from both the agent and the landscape model is required. In cases such as the crop harvest, the IM provides an optimal choice for managing crop values. The agent model encapsulates agent goals and behavior. Crops are not a part of either. For the agent, a crop is a means to an end — survival. Similarly, the notion of a crop does not belong in the landscape model that abstracts *natural* environment processes. Thus, a crop is truly a product of the agent model and landscape model interaction. The IM, having the capability to draw knowledge from both composed models, is the ideal place from which the state of the crops can be managed. It is through the explicit modeling of all four of these interactions, the usage of aggregation/disaggregation, timing control, and other mechanisms (described below), by which the entire composition complexity is managed.

It should be noted that the IM does not initiate interaction with a sub-system model on its own. It is a model of the interaction between the sub-system models and, as such, it sends data or a control message to a sub-system model as a result of input from the other. However, the IM must be cognizant of the differences in timing mechanisms and data content for each sub-system model. The IM must respond to input from both composed models and, in the case of the discrete-event agent model, may not know the exact timing of such inputs. Therefore, it is appropriate to use a discrete-event modeling formalism for the IM within the MedLand human-landscape model. With this in mind, the IM may potentially inject data into one of the sub-system models at any time. The time at which an event input is injected may not align with a regularly scheduled discrete-time instant. This poses a potential problem (see Huang [2008]) since the time-dependent functions within the

discrete-time landscape model must be executed at specific time steps. In the current model, the agent and landscape models operate on the same cycle. However, each agent sends independent control messages to interact with its environment. The IM then aggregates all agent input to a particular landscape model process. It then disaggregates the output of the process to provide each agent the portion of the landscape model result with which it is concerned.

Another sub-system model disparity that the exemplar IM accounts for is environmental resolution (Ullah et al. 2008). The agent manages cells with a 100 square meter resolution (10m $\times$ 10m). The landscape models use a finer granularity of 25 square meters (5m $\times$ 5m). Thus, each agent action impacts four landscape model cells. This has important implication during each of the view, farm, and harvest interactions. As stated above, the agent uses soil depth, and cover, and a distance cost from the village to determine the land value. During viewing, the IM takes the mean value of the soil depth and land cover from each of the four associated cells and provides this to the agent. The distance cost calculation is special. It is not just the length between two points; it is weighted by the type of terrain over which the agent must travel to get to that point. The distance cost is therefore a value created by an association between an agent model attribute (village location) and a landscape model attribute (terrain) and, as such, is an attribute of the IM itself. The IM used existing GRASS module functionality and scripts in order to calculate these mappings and distance costs.

When an agent specifies a farming action, the IM will disaggregate the impact from the agent's one "farmed" cell to the four representative cells in the landscape model. When an agent decides to harvest a crop, it receives back from that action, an amount of crop harvested as a yield. This yield is dependent upon the environmental factors in which it was

grown (soil depth, rainfall, etc.). The crop itself is neither part of the landscape model nor of the agent model. However, for this attribute of the hybrid model to be useful, it requires data from both sub-system models and is therefore well-suited to be formulated within the IM. Thus, it is the IM that intercepts an agent's harvest request, queries the landscape model for environmental values which impact crop yield, calculates the actual yield, and provides this value to the agent. In turn, the IM collects agent impact data (farming and village site data) and provides this to the landscape model in order to facilitate the erosion model dynamics. Agent impacts are another model characteristic which, while generated as a result of agent decisions, make little sense on their own in either the context of agent decision making or in pure landscape dynamics. So, again, this is another attribute of the hybrid model that is best suited within the interaction model.

As stated previously, the IM provides visibility into the hybrid model interactions and behaviors. Consider that the agent model uses a deterministic, rules-based approach to drive its basic behaviors. However, it also uses stochastic methods to resolve conflicts between agent farming locations. This leads to non-deterministic results when it comes to individual household land holdings and, ultimately, survival. The landscape model also uses deterministic processes to model the environmental behavior. However, the exact process for each sub-system is unknown. Researchers strive to include more details into the sub-system models in order to test their understanding of these real systems. Thus, the sub-system model algorithmic complexity has grown due to the addition of a large number of variables.

While the range of input variables and initial conditions can be *a priori* formulated when each sub-system model is considered independently, it becomes less so when the two models interact. The impact that each variable has on the hybrid system behavior can become very difficult to determine. This is exacerbated by the fact that the agents, working

with landscape data, exhibit emergent behaviors in farming practices such as farming location and land management (*i.e.*, farmed or fallowed). This, in turn, leads to different erosion patterns as a result of the agent's actions. By using an interaction model, the researchers are able to model and manage the data passing between the sub-system models by focusing on things such as boundary values and input/output relation (*i.e.*, sensitivity analysis) for a range values. Furthermore, having an IM reassures the modeler that only specific data is being passed between the sub-system models. This helps to ensure that as the internal complexity of each model rises in the form of algorithmic or deterministic complexity, the aggregate complexity remains controlled. Thus, if the emergent behavior is invalid, a much smaller subset of processes and variables need be examined and modified. Finally, another important factor is the scalability of the IM since the amount of data exchanges and the frequency of sub-system models' interactions can be separately handled.

As stated previously, GRASS models do not have an innate sense of timing. Timing was provided directly by the interaction model. In other words, it played the role of the CCATimer. This direct coupling between IM and timer model enabled more direct mappings as the IM built scripts dynamically which were run to create new maps that represented the environmental model I/O. The downside to this tightly coupled approach was managing the changes required when either the interaction or the environmental model interface changed. This became a key motivator in devising an approach for aeScape that abstracted the two tasks of timing control and interaction.

5.2.3.1 Interaction mappings

There are three main interactions for which mappings are required between the farmer model and the landscape model. The first is for farmer land assessment. This is the point at which the farmer agent looks around the landscape to decide where might be the best

place to farm. The second is when the farmer actually farms the land. This creates an impact on the environment that must be taken into account when the erosion model is run. The third mapping is required for when the farmer harvests the crop. As discussed previously, a crop is fully part of neither the farmer nor the landscape model.

Farmer land assessment requires an aggregation of landscape attributes into a single value that represents land quality to the farmer. These attributes are soil depth, soil quality, and distance from the farmer's village. An aggregation function transformed these three values into a single value within a specified range. The farmer agent was able to then value-order the viewed land to determine which offered the best prospect for a good crop. However, due to the disparity in the landscape resolution, as part of the interaction process the IM calculates an average of the attributes from four landscape cells for each aggregation function to yield the land value for a single farmer cell. Setting this to equation, let $\overline{s}_d$ represent the average soil depth of the four landscape cells, $\overline{s}_q$ represent average soil quality, $\overline{d}$ represent average distance from the farmer's village, and l_v be the land value. Then $f(\overline{s}_d, \overline{s}_q, \overline{d}) \mapsto l_v$ is the mapping equation. Of course, the previous equation is generic. Specific content of $f(\overline{s}_d, \overline{s}_q, \overline{d})$ is domain dependent. While the specifics of this MedLand mapping function changed over time, the most recent version was $l_v = l_{v(max)} \times (((\overline{s}_q \times \overline{s}_d)/(s_{q(max)} \times s_{d(max)})) - (\overline{d} \times d^*))$, where $l_{v(max)}, s_{q(max)}$, and $s_{d(max)}$ are the maximum land value, maximum soil quality, and maximum soil depth respectively; and d^* is the weighting factor for the distance.

The second mapping function accounts for farmer impacts on the environment. As described previously, farming reduces the land cover that prevents erosion from occurring. Thus, more erosion occurs, which reduces the soil depth. The erosion process, E, is a water transport function with input variables of rainfall, r, that specifies how much water

is available to move soil, and soil depth, d, that enumerates how much soil is available to move. The output of this function is then scaled by the product of two erosion factors — a land cover factor, c, and soil compaction factor, k. These scalar values would each range from $[0.0 - 1.0]$, where a value of 0.0 indicates all erosion is prevented. Let these sets of possible values be denoted C and K, respectively. The cumulative erosion effect would be represented by $E(r, d) \times c \times k$. For simplistic, rain-fed farming methods, the agents would not have the capability to affect rainfall or soil depth to any great extent. However, their actions can modify both the land cover and the soil compaction values. In order to farm the area, the agent removes land cover to a point at which crops can more easily be grown en masse. If the land cover value, k, represents the land cover values before farmer actions, $k^{'}$ may be used to represent the land cover values after farming activity. Then, $k^{'}$ is mapped to the land cover factor, c (*i.e.*, $f(k^{'}) \mapsto c$). The values of C range from $[0 - 1.0]$, while the values of K, the set of all k, were specified to be $[0 - 50]$. A linear function can be used to map K to C.

As discussed previously, a crop is an entity that exists within the environment but is managed by a farmer. Its existence is solely the result of an interaction between farmer and environment for the manner in which these two models have been created. Thus, for a farmer to garner food from the crop, it is natural that the interaction model be involved. A power regression formula $y = 43.951 \times (s_d^{0.522071})$ is used to calculate the yield, where y is the yield and s_d is the soil depth in centimeters (Sadras and Calvino 2001). This formula assumes a constant average rainfall amount across the landscape. Different formula can be calculated and applied for varying rainfall amounts. In summary, while crops are a farmer-managed resource and crop yield is a farmer entity, landscape data is required to assign it value.

5.2.4 MedLand model simulation

The MedLand models are designed to investigate the impacts of farming in the Early Neolithic and Bronze Age in the circum-Mediterranean region. (MEDLAND 2009). The environmental conditions of the model were set according to what is believed to be the conditions in southern Spain at 7000 BP: moderate Mediterranean rainfall regime, Mediterranean Terra Rosa soils, and an initial land cover of Mediterranean woodland. Data from ethnoarcheological investigations of similar modern farmers has been used to devise the agent model, while the simulation results are compared against archeological data (Mayer et al. 2006). With the simulation described in this paper, the MedLand research group sought to better understand the workings of the hybrid model and how it represented the underlying human and environment sub-systems. Questions that were focused on included:

1. Can the environment sustain a moderately sized village population using a known archeological site?

2. How much do certain factors contribute to population sustainment?

   (a) Soil depth (and associated erosion)

   (b) Labor force versus the household size and caloric requirements

   (c) The number of households

3. What impacts do the farmers have on the landscape?

4. What are the impacts of different data exchanges between agent and landscape (*e.g.*, selected data elements and data size/resolution) and control frequency of data exchanges?

5.2.4.1 Initial conditions

Two village sites were chosen to allow examination of how topography influences the outcome of the model. The location of the two simulated villages in our experiment can be seen in figure 5.3. One of these villages (Village 2 in the model) is located at a higher elevation, up in the drainage network on the relatively steep valley flank. The other (Village 1 in the model) is located much further down the drainage network in a broad flat area of the valley. The details of some of the major initial conditions are given below. All of these values may be set at the start of the simulation. The intent is to emphasize the number of variables in the hybrid model, many of which may have profound impacts on the simulation results and increase the difficulty of managing the model's complexity.

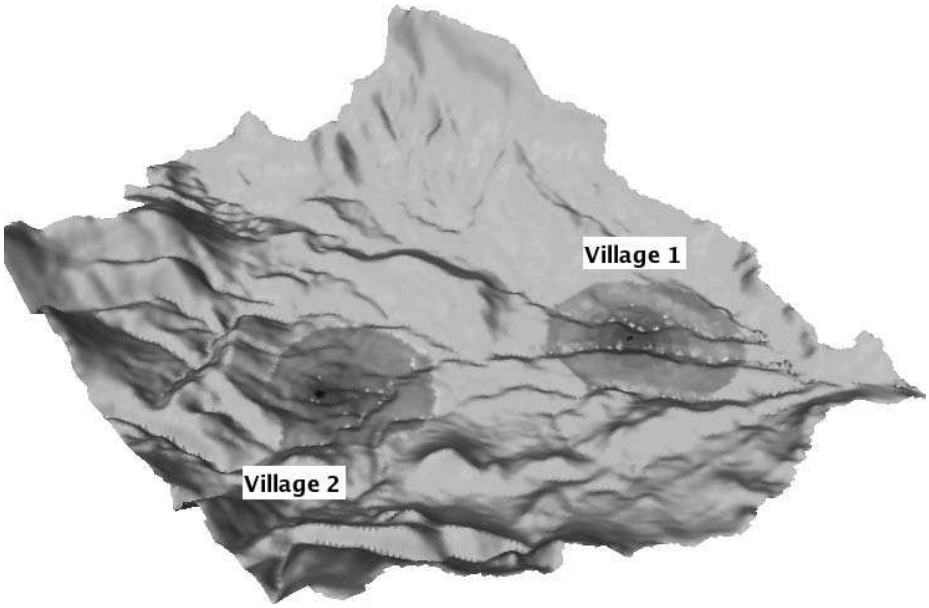

Fig. 5.3. Penaguila Valley with village locations and erosion impacts.

In the simulation, each village consists of 3 households, each with an initial population of 6 people. The initial birth rate is set at a 3% probability and the initial death rate is set at 2% (Cowgill 1975; Wood et al. 1998). In the event of a particularly good or bad

year (when yield versus need ratio is high or low, respectively), the birth rate changed at a rate of 1% per year while death rate would inversely change at 5% per year. The maximum birth and death rates were capped at 5% and 100% probability, respectively. The minimum birth and death rates were 0% and 2% probability, respectively.

Maximum yields of wheat and barley (in a soil depth greater than or equal to 100 cm) was set to 460 kg/Ha. Minimum yields (in a soil depth less than or equal to 13 cm) were 168 kg/Ha (Thomson et al. 1986). Yields for soils of depths between 13 cm and 100 cm were scaled by the power regression formula provided in section 5.2.3.1. Agents were set to initially expect a yield of 450 kg/Ha in the first year and, from then on, to expect the same yields as were actually produced in the prior year. As extensive grazing has not yet been added to the agent model, the impact of herd animals (goats and sheep) is only modeled by the amount of barley (grains, straw, and chaff) needed for supplemental herd feeding over the fall/winter months.

Each member of the household populace required 1,000,000 kilocalories of wheat per year and 250,000 kilocalories of milk and meat per year. Wheat was considered to be directly consumed in the form of bread and porridges, while barely was considered only as a fodder for herd animals that produced meat and milk products to be consumed by the households. Therefore, after harvest-loss, seed reserve-loss, and processing-loss, wheat provided agents with 3500 kcal/Ha, while barley only provided agents with 213.87 kcal/Ha (Thomson et al. 1986). In the simulation, only 50% of the population was able to produce labor and each productive person was considered able to produce 300 man-days per year. Wheat and barley farming required 50 man-days per Ha per year (Simms and Russell 1997).

In order to speed land patch querying and village-level land negotiations (currently the most time-costly operation in the agent model), villages in the current simulation were

given a "viewable" territory based on a 10 minute anisotropic walking-time cost-radius from the village center, which translates into an area of about 160 Ha. Village area was calculated using the population density formula, $A = V_p/D$, at the start of the simulation and every time the village population changed. A is the area occupied by the village in square meters, V_p is the current village population, and D is the population density coefficient of people per square meter. The current simulation used a D value of 0.0159.

Due to the many parameters that can be set when the models are initialized, a graphical interface was devised that could assist the modeler. This interface, some of which is shown in figures 5.4 and 5.5, allows the modeler to set variable values for agent and agent-landscape interactions. Figure 5.4 shows the various settings available for the household agents. This includes settings for birth and death rates, how much food each person requires, and how much labor each member of a household contributes. Figure 5.5 is a view of the Environment Model settings. Here, the GRASS environment can be specified by values such as the mapset to use, and what digital elevation map (DEM) should be used for defining the region and initial elevation values.

5.2.4.2 Results

The simulation was run for 200 cycles (at a temporal resolution of one cycle per year), and finished in less than 30 hours. The extent of the agents' impacts on their environment was assessed by comparison with a control landscape model. This control model is run with the same initial conditions used in the hybrid agent-landscape simulation run. The results of the control model simulate the amounts of erosion and deposition that would have taken place if no people inhabited the landscape.

The simulation results provided some clear answers to the questions posed in section 5.2.4.1. Realistic values, derived from ethnographic data, archeological data, and other re-

search studies, were used. The simulation results revealed hybrid model dynamics that were more quantifiable, and that could be directly attributed to complex interactions between the individual household and landscape models. The domain experts are able to understand and explain model composition complexities that previously could not be examined at such a level of detail and scale for the agent and landscape models.

1. The environment *can* sustain a moderately sized village population using known archeological sites. However, the extent to which the population is successful at surviving is dependent upon the output of both the conflict resolution and population growth (*i.e.*, birth and death) stochastic processes.

2. How much do certain factors contribute to population sustainment?

   (a) Soil depth (and associated erosion) only plays a significant role if the depth falls within the minimum and maximum values for yield. If soil depth is beyond the maximum for yield, the household does not notice any changes in soil depth with relation to its yield. If below the minimum value, the household ignores the land.

   (b) In examining labor force versus the household size and caloric requirements, careful consideration must be given to the initial values of the variables. The wrong values either provide too easy of a scenario for the households or one in which it is impossible to survive.

   (c) An impact of changing the number of households is the change in the distance from the household to the farmland. This is because as the number of households increases, some households had to travel further to find viable land for farming. Additionally, due to the stochastic nature of how households were assigned management rights to a land area, some unlucky households were left with farmland

that produced lower yields than average and, therefore, their populations did not do as well. With these impacts, the rules used to model how households determine where to farm (*e.g.*, soil quality and distance) could be more thoroughly examined, analyzed, and compared to anthropological data.

3. Farming is shown to increase landscape erosion. However, village sites, which are considered to be compacted soil, decreases erosion. The extent of these impacts is dependent upon the environmental conditions surrounding those areas. For example, if an area would naturally incur deposition of soil, then farming may just cause the soil depth to be near static. On the other hand, an area already prone to erosion would decline rapidly, forcing the agent to soon give up its farm land and seek better farming areas.

4. The different data exchanges between the agent and landscape sub-system models show the importance of maintaining their separation via an interaction model. An important idea is that each model need not expose all of its data to its external environment. It provides the data it wants to the IM and the IM may then manipulate that data as needed to conform to the input of the other sub-system model. This reduces the complexity within each sub-system model. Furthermore, with respect to frequency of data exchanges, each model may use its own timescale. The IM may then manage data transformation and exchange frequency at the appropriate times. Reducing data exchanges can improve simulation execution time (performance) since interaction between models can be controlled both in quantity and frequency. For our development environment (*i.e.*, DEVSJAVA), aggregating multiple, similar exchanges

(households desiring to examine the same land, for example) proved an efficient way to improve performance.

In answering each of these, examining the data exchanged between the agent and landscape models was key. There were many factors to consider in determining, for example, why a population decreased. By examining the interaction between the two sub-system models, it might be seen that yield values were plentiful for supporting the household's population. This then leads the researcher to examine the internal processes of the agent model, such as the amount of land that could be farmed and the stochastic population growth. For example, even when the average birth rate is higher than the average death rate, it is possible that random number generation will produce more deaths than births. Alternatively, it might be found that yield values would not support the population, even after the household tried farming many different lands. This then leads the researcher to examine the landscape soil depths to determine why they became so low. In short, it is the formulation of the interactions and their visibility that leads the researcher along a logical path to understanding why the hybrid model is behaving the way it is.

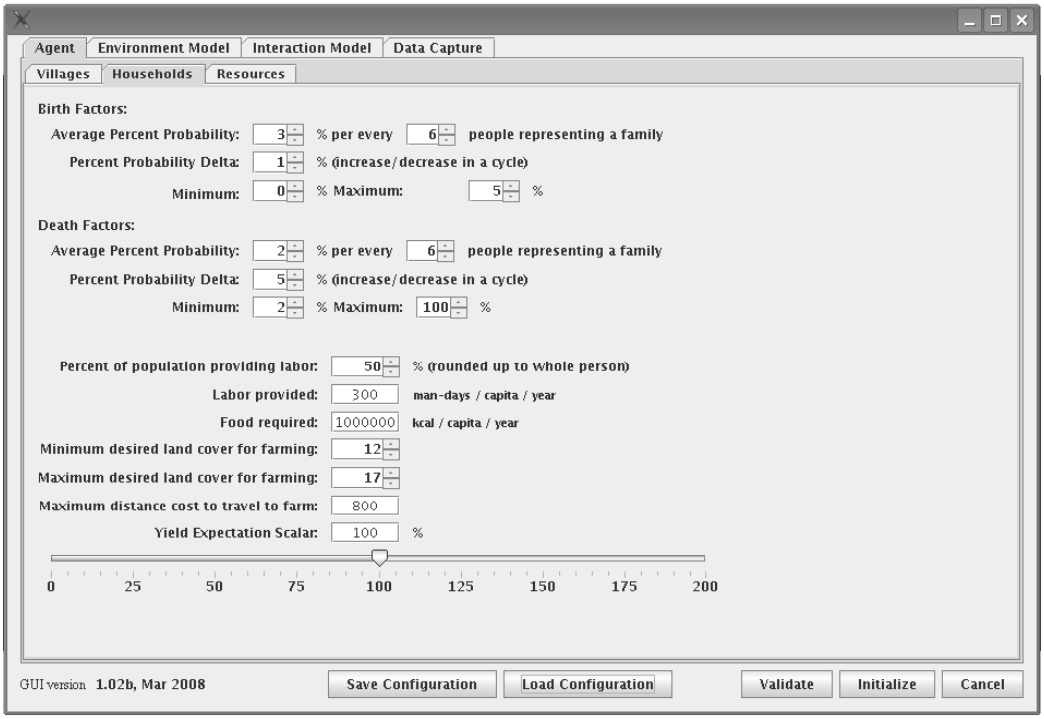

Fig. 5.4. MedLand initialization interface — Agent Model:Household tab shown.

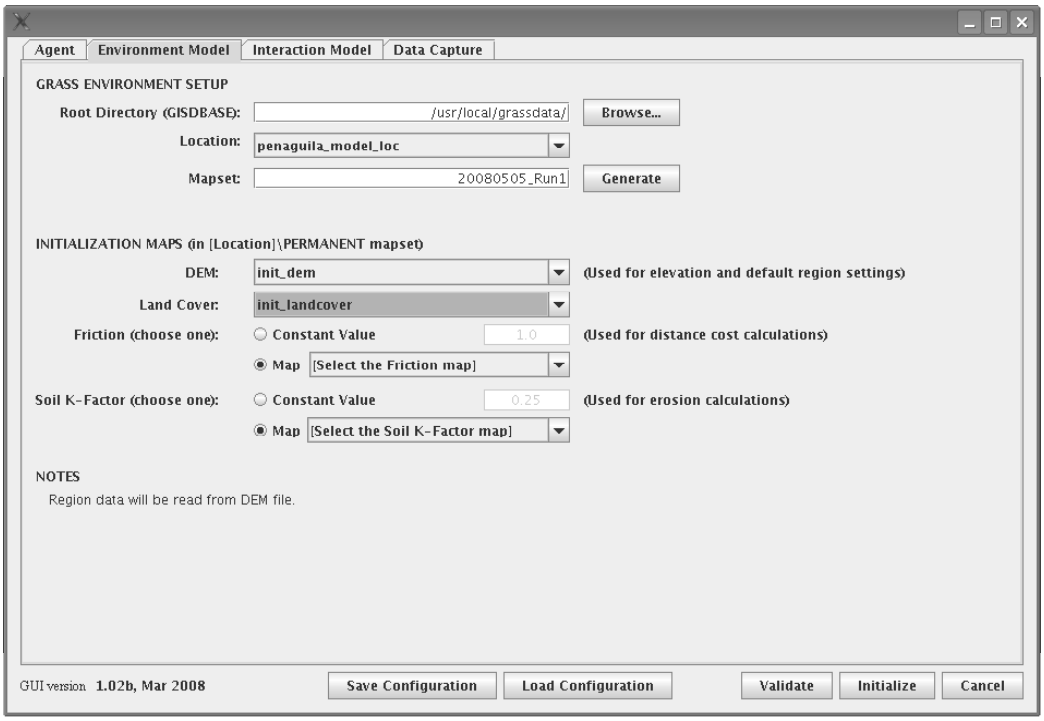

Fig. 5.5. MedLand initialization interface — Environment Model tab shown.

CHAPTER 6

CONCLUSION AND FUTURE WORK

A novel approach to composing a hybrid model derived from the Discrete Event System (DEVS) specification and another from the Composable Cellular Automata (CCA) specification has been proposed by this dissertation. The approach uses the Knowledge Interchange Broker (KIB) concept of composing models from formalism through realization of the two models. This is done using a third model, an interaction model, that explicitly models the interactions between the composed models.

The DEVS specification is an existing specification that can describe discrete and continuous time models. The CCA specification was devised as part of the research for this dissertation. The CCA specification is derived from a multi-component, discrete-time system specification. The CCA describes a formal method by which input and output can be managed from a cellular automata (CA). This formal method of I/O is lacking from current CA specifications. Without the formal I/O capability, the CA specification could not be used as part of a poly-formalism composition.

In addition to the CCA specification itself, an approach was presented that allowed the CCA specification to be implemented in the Geographical Resources Analysis Support System (GRASS) development environment. One difficulty in building a model in GRASS that is to be composed with a DEVS model is that GRASS has no innate concept of time. Thus, a DEVS timer was devised that would provide timing control for the CCA dynamics when the CCA was implemented in an untimed environment.

6.1 Conclusions

The KIB concept can be used to define a composition between a DEVS model and a CCA. It has been described and demonstrated in this dissertation how an interaction (IM) model that formally composes the two models can be devised. The IM in this composi-

tion accounts for the disparities that exist between the two model specifications. These specification disparities include differences in structure (hierarchical versus network), and timing (discrete-time versus discrete-event). The approach to devising an interaction model that manages the disparities in the realization of both models was described as well. This approach included the concept of modeling behavior and attributes that are unsuitable for either of the agent or environment model due to these items existing as a product of the interaction. These interaction products that are best suited to be implemented within the IM, should help enforce the idea that the IM is a model of the interaction.

An agent-environment exemplar model was developed for this dissertation and successfully demonstrated the feasibility and utility of the composition approach. This exemplar model, called aeScape, is based on the well-known Sugarscape module developed by Epstein and Axtell (Epstein and Axtell 1996). In aeScape, the agents are created to be stand-alone models and the model interactions are made intentionally more complex to exemplify the model composition via the interaction model, and to demonstrate the CCA structure and dynamics. The aeScape agent is developed in DEVSJAVA, a Java implementation of the DEVS specification, and the CCA is developed using GRASS. At this level of the composition, domain knowledge and domain representation within each model plays a much larger role in determining the disparities between the two models to be composed. Data type (object constructs versus primitive data types stored within files) is one example, and devising an interface between the Java Runtime Environment (JRE) and a GRASS C module, is another.

In addition to aeScape, this dissertation described how the interaction model and other KIB concepts were applied with novel ideas and for practical use in the MedLand project. Like aeScape, MedLand required composition of agent and environment models.

Also like aeScape, the agent and environment models are developed in DEVS and GRASS, respectively. Where MedLand differed from aeScape was in the testbed it offered as a result of having a domain with real-world needs and data. The MedLand IM was devised around an abstraction of the interaction between the environment and agent models, within a concrete domain. The simulation data could therefore be tested against ethnographic and anthropological data in order to assess the credibility of the output.

In examination of aeScape and MedLand, two levels of generality can be conceived. The first is at the model specification level. Any two composed models, one using the DEVS specification and the other using the CCA specification, will comply with the theoretical foundation put forth in chapter 3. An interaction model created for composing these two models will have to account for the theoretical differences between these specifications. The second level of generality is provided at the domain level. The aeScape hybrid model and the MedLand hybrid model employ very different interaction models. Though both are agent-environment applications, the specific domain application is different. Thus, the IM specifics are also different. That being said, numerous changes can be made to the aeScape models without requiring significant changes to the aeScape IM. A similar statement can be made about the MedLand IM. As long as the composed models retain the same domain abstractions, then they will continue to interact in a similar manner and their IM will remain a valid abstraction of their interaction.

Through model development, revision, and simulation, the KIB concept, and IM specifically, proved itself to be a benefit in hybrid model management. During development, the loosely coupled model architecture allowed the sub-system models to be developed independently. Each could be created and separately tested. Then, pieces could be added together and tested in a modular fashion. As revisions were made, the sub-system models

could be modified independently with minimal impact on the other models in the hybrid system model. Lastly, the IM provided a central point from which to draw data about the model interaction; data that would require knowledge of multiple sub-system models and may have required some effort to gather and compile the data externally.

6.2 Applicability

The results of this research extend the potential application of modeling and simulation in computer science and, more broadly, computational sciences. In light of the increased usage of hybrid simulation models, this research provides a formal approach to composing discrete-event rules-based agent and discrete-event cellular-automata environment models. It adds another tool from which a modeler may choose. The benefits that this approach offers is reduced modeler workload for verification and validation of the models through the use of proven model specifications, and accounting for both formalism and realization disparities in the composition. Other benefits of this approach are interaction complexity management through visibility of the exchange between models; and a clear separation between models. The latter releases the composed models from the burden of having intricate knowledge of the other, thereby improving reuse and reducing the impact on the hybrid system model from changes to a single model.

The efforts of this dissertation may be used to guide modelers from any number of different fields of study in creating their own hybrid model. First, the IM devised for the MedLand project is functional and can be used to work toward meeting MedLand goals. It can be expanded upon to embrace further model details occurring in the interaction between human and landscape models. Alternatively, the aeScape model can continue to be used to explore the composition approach and elaborate on other efforts within Sugarscape-like worlds. Second, the IMs within this dissertation may be expanded upon, or a new model

devised, that requires composition of a DEVSJAVA implementation of a DEVS agent-based model and a GRASS implementation of a CCA. In this scenario, the disparities accounted for discussed herein are accounted for if applicable to the new domain and most of the hybrid model can likely be reused; with the most likely exception be reformulation of specific behaviors. In other words, the details discussed in sections 3.2.1, 3.2.2, and 3.2.3 remain applicable. At the next level, a modeler could use this dissertation, and the details provided in section 3.2.1 and parts of section 3.2.3, to devise a composition between two entirely different DEVS and CCA models. Lastly, this dissertation may be used simply as a guide for a modeler to devise a new composition between two entirely different specifications. If this dissertation is used as a guideline, then the modeler is reminded to begin with two specifications, each of which clearly details the structure of the model to be developed, the dynamics of the model, and how that model interacts with external models.

6.3 Future Work

A number of different research paths are possible when starting with the foundation provided in this dissertation. These are provided below. While in no specific order, the topics tend to be listed from more domain-specific, to more theoretical.

6.3.1 Domain-specific approach

aeScape provided a *very* small portion of a domain-specific modeling library. The difficulty with a very general, theoretical specification like DEVS is that it may come across as unaccessible to domain experts who might not see DEVS as easy to use as other, domain-specific tools. Thus, to make DEVS more accessible to this audience so that they too can gain the benefits of its theoretical underpinnings, it should be possible to provide a domain library that conforms to the DEVS specification but that provides concepts with which a domain expert is familiar. Given the almost infinite ways in which a domain could be

defined, this is a very large undertaking. By its very nature, this library would have to define a hierarchy of domains and choose a subset to which it would apply. But, having this library, or some tool that makes DEVS more accessible, would make multidisciplinary modeling and simulation projects like MedLand easier to conduct.

With a domain-specific library in place, or even a collection of them, the development framework could be made more accessible to other modelers through common platforms such as Java Web Start. The more narrow focus of behavior and interaction could provide an opportunity to create an interface specialized to a particular library. The obvious downside, as elaborated on early in this dissertation, is that the library would narrow the applicability of the tool. So, to garner maximum benefit from the labor required to create the library, it should be generated such that it is usable by as large a community as possible.

6.3.2 Visualization

As stated previously, in addition to generating the interaction data maps, the IM is also a convenient model from which to generate data visualizations. Using GRASS methods, the IM can convert sugar and landscape state attributes into color gradients representing the state of the environment model. Then, it can overlay agent location data onto that map. Furthermore, the agent colors can also be manipulated to provide a general assessment of the sugar wealth of individual fiends. Finally, these color-managed maps can be output as images, and stepped through in order to provide an animation that represents the agent and environmental state changes throughout the simulation. Alternatively, a visualization framework entirely external to both models can be used. While some data capture was employed, more formal methods could be added to the framework, which would then allow for formal visualization techniques.

6.3.3 Robotic agents

An area of personal interest to the author is to expand the application of this composition approach, between a DEVS agent and a CCA environment, into the physical realm. This can be achieved by encompassing robotic applications into the agent model. Thus, the environment model would detail the environment in which the robot operates. Given the growing robotic applications, this may have benefit for studying robotic system, and developing robotic applications.

6.3.4 Distributed and parallel execution

A possible next step for this research is in the use of distributed and parallel execution. Given that an interaction model provides the capability to develop the composed models independently, a logical next step may be to allow the models to remain distributed in their execution and operate in parallel. The IM will incur the additional responsibilities of synchronizing the execution, and managing the model communication network. This research would also lend itself to future work in interoperability (see section 6.3.7).

6.3.5 Visual CCA modeling

There is a cellular automata visual modeling effort that was developed concurrently with this dissertation (see Sarkar [2009]). This thesis effort provided the capability to visually model CA through a graphical user interface. The modeler is capable of specifying cell attributes once and having them apply to the whole CA or a subset thereof. The model is also persisted in a database for ease of reuse. As it is an initial effort, the thesis makes use of a CA specification that does not account for external I/O. Thus, it is a stand-alone model. It would be beneficial to elaborate on both the thesis work, and the work of this dissertation by incorporating the CCA specification into the visual framework. In addition

to making the CCA easier to model, it would allow the modeler to devise other models, or multiple CCA in a visual environment.

6.3.6 CCA specification extension

As cellular automata are natural approach to modeling realistic environments, it may be necessary to model the environment at different resolutions. For example, one are may require a higher resolution or the modeler may wish to have the capability increase the granularity of the CCA tessellation. These approaches require that two CCA representing the same underlying system, have some kind of defined relationship between themselves, and between other CCA to which both may connect. Therefore, it may be useful to define a CCA hierarchy that allows this to occur. This hierarchy may have the added benefit of allowing a set of CCA to be treated as a single system from the perspective of a non-CCA external system, such as an agent model.

6.3.7 Framework extension to include interoperability

The results of this dissertation may be applied to on-going interoperability research — how two different simulations may be made to work together. The efforts discussed within this dissertation have all focused on composition — how two models within a simulation might be made to interact. Current interoperability research focuses on the simulator and treats the models it is simulating more as black box components — only the I/O is important. Some of this research provides time management and other kinds of support for the I/O. With the importance that domain plays in the composition between models, domain may also have a significant impact on interoperability. If nothing else, it may be wise to demonstrate that composition has a role to play in defining interoperability.

References

Alessa, L. N., M. Laituri, and M. Barton (2006). An "all hands" call to the social science community: Establishing a community framework for complexity modeling using agent based models and cyberinfrastructure. *Journal of Artificial Societies and Social Simulation 9*(4), 6.

Anderson, J. A. (2006). *Automata Theory with Modern Applications*. New York, New York: Cambridge University Press.

Barile, M. and E. W. Weisstein (2008). Neighborhood. http://mathworld.wolfram.com/Neighborhood.html (accessed September 2008).

Booch, G., R. A. Maksimchuk, M. W. Engle, B. J. Young, C. J., and K. A. Houston (2007). *Object-Oriented Analysis and Design with Applications* (3 ed.). Addison-Wesley Professional.

Braun, J., A. M. Heimsath, and J. Chappell (2001). Sediment transport mechanisms on soil-mantled hillslopes. *Geology 29*(8), 683–686.

Briesen, M. and J. R. Weimar (2001). Distributed simulation environment for coupled cellular automata in Java. In *Java, Parco*.

Bryson, R. and K. McEnaney-DeWall (2007). An introduction to the archaeoclimatology macrophsical climate model.

CoSMoS (2009). Component-based System Modeling and Simulation. http://sourceforge.net/projects/cosmosim (accessed June 2009).

Cowgill, G. L. (1975). On causes and consequences of ancient and modern population changes. *American Anthropologist 77*(3), 505–525.

Davis, P. and R. Anderson (2004). *Improving the Composability of Department of Defense Models and Simulations*. Santa Monica, CA: RAND.

de Lara, J., H. Vangheluwe, and M. Alfonseca (2004). Meta-modelling and graph grammars for multi-paradigm modelling in AToM3. *Software and Systems Modeling 3*(3), 194–209.

DeMers, M. N. (2002). *GIS Modeling in Raster*. New York, New York: John Wiley & Sons, Inc.

DEVS-Suite (2009). Parallel DEVS simulator. http://sourceforge.net/projects/devs-suitesim/ (accessed June 2009).

Dietrich, W. E., D. G. Bellugi, L. S. Sklar, J. D. Stock, A. M. Heimsath, and J. J. Roering (2003). Geomorphic transport laws for predicting landscape form and dynamics. *Geophysical Monograph 135*, 103–132.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong (2003). Taming heterogeneity–the ptolemy approach. In *Proceedings of the IEEE*, pp. 127–144.

El Yacoubi, S., B. Chopard, and S. Bandini (Eds.) (2006, September). *Cellular Automata*, Perpignan, France. 7th International Conference on Cellular Automata for Research and Industry, ACRI 2006: Springer. Proceedings. LNCS 4173.

Elamvazhuthi, V. (2008, April). Visual component-based system modeling with automated simulation data collection and observation. Master's thesis, Arizona State University, Tempe, AZ.

Epstein, J. M. and R. Axtell (1996). *Growing Artificial Societies: Social Science from the Bottom Up*. Cambridge, Massachusetts: MIT Press.

Fishwick, P. (1995). *Simulation Model Design and Execution: Building Digital Worlds*. Upper Saddle River, NJ: Prentice Hall.

Gibbs, J. (2009, April). An approach to evaluating computer network simulation tool support for verification and validation. Master's thesis, Arizona State University, Tempe, AZ.

Godding, G., H. S. Sajoughian, and K. Kempf (2004). Multi-formalism modeling approach for semiconductor supply/demand networks. In *Proceedings of Winter Simulation Conference*, Washington, D.C., pp. 232–239.

GRASS (2009). Geographic Resources Analysis Support System. http://grass.itc.it/ (accessed March 2009).

Hall, S. (2005). Learning in a complex adaptive system for ISR resource management. In *Spring Simulation Conference*, pp. 5–12.

Hancock, G. R. (2004). Modelling soil erosion on the catchment and landscape scale using landscape evolution models a probabilistic approach using digital elevation model error. In *Super Soil, 3rd Australian New Zealand Soils Conference*.

Hoekstra, A. G., J.-L. Falcone, A. Caiazzo, and B. Chopard (2008). Multi-scale modeling with cellular automata: The complex automata approach. *Eighth International COnference on Cellular Automata for Research and Industry (ACRI) 2008 LNCS 5191*, 192–199.

Hoekstra, A. G., E. Lorenz, J.-L. Falcone, and B. Chopard (2007). Towards a complex automata framework for multi-scale modeling: Formalism and the scale separation map. *International Conference on Computational Science 2007 LNCS 4487*, 922–930.

Huang, D. (2008, April). *Composable Modeling and Distributed Simulation Framework for Discrete Supply-Chain Systems with Predictive Control*. Ph. D. thesis, Arizona State University, Tempe, AZ.

IEEE (2000). HLA framework and rules. IEEE 1516–2000.

Ilachinski, A. (2001). *Cellular Automata: A Discrete Universe*. New Jersey: World Scientific.

Karsai, G., M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits (2004). Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology 12*(2), 263–278.

Kim, S. (2008, December). Simulator for service-based software system: Design and implementation with DEVS-Suite. Master's thesis, Arizona State University, Tempe, AZ.

Kim, S., H. S. Sajoughian, and V. Elamvazhuthi (2009, March). DEVS-Suite: A simulator for visual experimentation and behavior monitoring. In *High Performance Computing and Simulation Symposium, Proceedings of the Spring Simulation Conference*, San Diego, CA.

Kincaid, C. A., S. P. Mohanty, A. R. Mikler, E. Kougianos, and B. Parker (2006). A high performance asic for cellular automata (CA) applications. In *International Conference on Information Technology*, Los Alamitos, CA, USA, pp. 289–290. IEEE Computer Society.

Manson, S. (2001). Simplifying complexity: a review of complexity theory. *GEOFORUM 32*, 405–414.

Mayer, G. R. and H. S. Sarjoughian (2007, March). Complexities of simulating a hybrid agent-landscape model using multi-formalism composability. In *Proceedings of the 2007 Spring Simulation Conference (SpringSim '07)*, Norfolk, Virginia, pp. 161–168. IEEE Press.

Mayer, G. R. and H. S. Sarjoughian (2009). Composable cellular automata. *Simulation: Transactions of the Society for Modeling and Simulation International*. (in print).

Mayer, G. R., H. S. Sarjoughian, E. K. Allen, S. E. Falconer, and C. M. Barton (2006, April). Simulation modeling for human community and agricultural landuse. In *Proceedings of the 2006 Spring Simulation Conference (SpringSim '06)*, Huntsville, Alabama, pp. 65–72. IEEE Press.

MEDLAND (2009). Mediterranean Landscape Dynamics. http://www.asu.edu/clas/shesc/projects/medland/ (accessed March 2009).

Miller, A., C. M. Barton, S. Schmich, and S. McClure (2008). *Climate change and socioecological dynamics in eastern Spain*. Vancouver, BC: Invited symposium paper presented at the *73rd Annual Meeting of the Society for American Archaeology*.

Minar, M., R. Burkhart, C. Langton, and M. Askenazy (1996). *The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations*. Santa Fe, NM: Santa Fe Institute.

Mitasova, H. and L. Mitas (1993). Interpolation by regularized splines with tension: Theory and implementation. *Mathematical Geology 25*, 641–655.

Mosterman, P. and H. Vangheluwe (2004). Computer automated multi-paradigm modeling: An introduction. *Simulation Transactions 80*, 433–450.

Muzy, A., E. Innocenti, D. R. C. Hill, A. Aiello, J. F. Santucci, and P. A. Santoni (2004). Dynamic structure cellular automata in a fire spreading application. In *First International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, Setubal, Portugal, pp. 143–151.

Neteler, M. and H. Mitasova (2004). *Open Source GIS: A GRASS GIS Approach* (2 ed.). New York, New York: Springer Science + Business Media, Inc.

North, M., T. Collier, and J. Vos (2006). Experiences creating three implementations of the repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation 16*(1), 1–25.

North, M., T. Howe, N. Collier, and J. Vos (2005, October). The Repast Symphony development environment. In L. F. Perrone, F. P. Wieland, J. Liu, G. Lawson, M. Nicol, and R. M. Fujimoto (Eds.), *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, Chicago, Illinois, pp. 159–166. Argonne National Laboratory.

Ntaimo, L., B. Zeigler, M. Vasconcelos, and B. Khargharia (2004). Forest fire spread and suppression in DEVS. *Simulation Transactions 80*(10), 479–500.

Pardo, F. and L. Gil (2005). The impact of traditional land use on woodlands: a case study in the spanish central system. *Journal of Historical Geography 31*(3), 390–408.

Parker, D., S. Manson, M. Janssen, M. Hoffmann, and P. Deadman (2003). Multi-agent systems for the simulation of land-use and land-cover change: A review. *Annals of the Association of American Geographers 93*(2), 314–337.

Pilone, D. and N. Pitman (2005). *UML 2.0 in a Nutshell: A Desktop Quick Reference*. Sebastopol, CA: O'Reilly Media, Inc.

Sadras, V. and P. Calvino (2001). Quantification of grain yield response to soil depth in soybean, maize, sunflower, and wheat. *Agronomy Journal 93*, 577–583.

Sarjoughian, H. S. (2006, December). Model composability. In L. F. Perrone, F. P. Wieland, J. Liu, G. Lawson, M. Nicol, and R. M. Fujimoto (Eds.), *Proceedings of the 2006 Winter Simulation Conference (WinterSim '06)*, Monterey, California, pp. 149–158. IEEE Press.

Sarjoughian, H. S. and V. Elamvazhuthi (2009, March). CoSMoS: A visual environment for component-based modeling, experimental design, and simulation. In *Proceedings of the International Conference on Simulation Tools and Techniques*, Rome, Italy.

Sarjoughian, H. S. and D. Huang (2005). A multi-formalism modeling composability framework: Agent and discrete-event models. In *The 9th IEEE International Symposium on Distributed Simulation and Real Time Applications*, pp. 249–256.

Sarjoughian, H. S. and J. Plummer (2002). Design and implementation of a bridge between RAP and DEVS. Technical report, Computer Science and Engineering, Arizona State University.

Sarjoughian, H. S. and B. P. Zeigler (2000). DEVS and HLA: Complementary paradigms for modeling and simulation? *Transactions of the Society for Modeling and Simulation International 17*(2), 187–197.

Sarkar, S. (2009, April). An approach to visual modeling of cellular automata. Master's thesis, Arizona State University, Tempe, AZ.

Severance, F. L. (2001). *System Modeling and Simulation: An Introduction*. New York, NY: John Wiley & Sons Ltd.

Shiyuan, H. and L. Deren (2004, June). Vector cellular automata based geographical entity. In *Proceedings of the 12th International Conference on Geoinformatics – Geospatial Information Research: Bridging the Pacific and Atlantic*, University of Gävle, Sweden, pp. 249–256.

Simms, S. R. and K. W. Russell (1997). Bedouin hand harvesting of wheat and barley: Implications for early cultivation in Southwestern Asia. *Current Anthropology 38*(4), 696–702.

Sloot, P. M. A., B. Chopard, and A. G. Hoekstra (Eds.) (2004, October). *Cellular Automata*, Amsterdam, The Netherlands. 6th International Conference on Cellular Automata for Research and Industry, ACRI 2004: Springer. Proceedings. LNCS 3305.

Soto, M., P. Fall, M. Barton, S. Falconer, H. Sarjoughian, and R. Arrowsmith (2007, March). Land cover change in the Southern Levant: 1973 to 2003. Presented at ASPRS Southwest technical conference.

Thomson, E. F., F. Bahhady, A. Termanini, and M. Mokrebel (1986). Availability of home-produced wheat, milk products and meat to sheep-owning families at the cultivated margin of the NW Syrian steppe. *Ecology of food and nutrition 19*, 113–21.

Tisue, S. and U. Wilensky (2004). Netlogo: Design and implementation of a multi-agent modeling environment. In *Proceedings of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence*.

Tomlin, C. D. (1990). *Geographic Information Systems and Cartographic Modeling*. Englewood Cliffs, New Jersey: Prentice Hall.

Ullah, I. and C. M. Barton (2007, April). *Alternative futures of the past: modeling Neolithic landuse and its consequences in the ancient Mediterranean*. Austin, Texas: Contributed paper presented at the *72nd Annual Meeting of the Society for American Archaeology*.

Ullah, I., G. Mayer, M. Barton, H. Sarjoughian, and E. DiMaggio (2008). Ancient Mediterranean landscape dynamics: Hybrid agent and goespatial models as a new approach to socioecological simulation. In *SAA Symposium*, Vancouver, BC, Canada.

Wainer, G. (2006). Applying Cell-DEVS methodology for modeling the environment. *Simulation Transactions 82*(10), 635–660.

Wainer, G. A. and N. Giambiasi (1997). Specification, modelling and simulation of timed Cell-DEVS spaces. Technical Report 97-006, Informe Técnico, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina.

Warren, S. D., H. Mitasova, M. G. Hohmann, S. Landsberger, F. Y. Skander, T. S. Ruzycki, and G. M. Senseman (2005). Validation of a 3-d enhancement of the universal soil loss equation for preediction of soil erosion and sediment deposition. *Catena 64*, 281–296.

Weisstein, E. W. (2008). Cellular automaton. from mathworld — a wolfram web resource. http://mathworld.wolfram.com/CellularAutomaton.html (accessed September 2008).

Willgoose, G. (2005). Mathematical modeling of whole landscape evolution. *Annual Review of Earth and Planetary Sciences 33*(1), 443–459.

Wolfram, S. (1983, July). Statistical mechanics of cellular automata. *Reviews of Modern Physics 55*, 601–644.

Wolfram, S. (2002). *A New Kind of Science*. Champaign, Illinois: Wolfram Media, Inc.

Wood, J. W., G. L. Cowgill, R. E. Dewar, N. Howell, L. W. Konigsberg, J. H. Littleton, R. D. Attenborough, and A. C. Swedlund (1998). A theory of preindustrial population dynamics: Demography, economy, and well-being in Malthusian systems [and comments and reply]. *Current Anthropology 39*(1), 99–135.

Zeigler, B., H. Praehofer, and T. Kim (2000). *Theory of Modeling and Simulation: Integrated Discrete Event and Continuous Complex Dynamic Systems* (2 ed.). San Diego, California: Academic Press.

Zeigler, B. P. (2006). DEVS&DESS in DEVS. In *DEVS Integrative Modeling & Simulation Symposium*, Huntsville, Alabama.

APPENDIX A

DEVS SPECIFICATION

The parallel DEVS formalism describes an atomic model component, M, as an octuplet:

$$M = \left\langle X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \right\rangle, \text{where} \tag{A.1}$$

$X_M = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values,

$Y_M = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values,

S is the set of sequential states,

$\delta_{ext} : Q \times {X^b}_M \rightarrow S$ is the external state transition function,

$\delta_{int} : S \rightarrow S$ is the internal state transition function,

$\delta_{con} : Q \times {X^b}_M \rightarrow S$ is the confluent transition function,

$\lambda : S \rightarrow Y^b$ is the output function,

$ta : S \rightarrow \Re^+_{0,\infty}$ is the time advance function, and

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the set of total states.

Note that the superscript 'b' refers to a "bag" or set with possible multiple occurrences of its elements.

The parallel DEVS formalism describes a coupled model component, N, using a septuplet:

$$N = \left\langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC \right\rangle, \text{where} \tag{A.2}$$

$X_M = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of input ports and values,

$Y_M = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of output ports and values,

D is the set of component names,

M_d are the DEVS model components, $d \in D$,

$EIC \subseteq \{((N, ip_N), (d, ip_d)) | ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$ is the external input coupling that connects external input ports (from the coupled model) to the component inputs,

$EOC \subseteq \{((d, op_d), (N, op_N)) | op_N \in OPorts, d \in D, op_d \in OPorts_d\}$ is the external output coupling that connects component outputs to (coupled model) external output ports,

$IC \subseteq \{((a, op_a), (b, ip_b)) | a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$ is the internal coupling that connects the output of component a to the input of component b. Note that no direct feedback loops are allowed (*i.e.*, $((d, op_d), (e, ip_e)) \in IC \Rightarrow d \neq e$).

APPENDIX B

CCA SPECIFICATION

A Composable Cellular Automata (CCA) extends the concept of a Cellular Automata (CA) by explicitly accounting for input to and output from the cellular network from and to an external system. A CCA's structure and dynamics can be described as a network, N_{CCA}, of homogeneous components (cells), M, using the following specification:

$$N_{CCA} = \left\langle X_N, Y_N, D, \{M_{ijk}\}, T, F \right\rangle, \text{ where} \tag{B.1}$$

$X_N = \{ \overline{X}_{ijk} \,|\, (i,j,k) \in D \} \vee \emptyset$ is the set of external input mapped to the network, N_{CCA},

$Y_N = \{ \overline{Y}_{ijk} \,|\, (i,j,k) \in D \} \vee \emptyset$ is the set of external output for $\{M_{ijk}\}$ from the network, N_{CCA},

$T = \{ h_m \,|\, 0 \leq m \leq n \}$ is the time-ordered, finite set of time intervals, h_m, (*i.e.*, $\{h_0, h_1, h_2, \ldots, h_n\}$) such that $m, n \in \mathbb{N}_0, \mathbb{N}_0 \equiv (\mathbb{N} \cup \{0\}) - \{\infty\}, h_m \in \mathbb{N}^*$, $\mathbb{N}^* \equiv \mathbb{N} - \{0, \infty\}$ and $\forall h_m$, h_m occurs before h_{m+1} (*i.e.*, h_0 occurs before h_1, h_1 occurs before h_2, etc.),

$F = \{ f_{out}, f_{in} \}$ is the set of I/O mapping functions between the network and its cell components, $\{M_{ijk}\}$, where $f_{out} : \bigcup_{(i,j,k) \in D} \overline{Y}_{ijk} \mapsto Y_N$ and $f_{in} : X_N \mapsto \bigcup_{(i,j,k) \in D} \overline{X}_{ijk}$,

$D = \{ (i,j,k) \,|\, a \leq i \leq b, c \leq j \leq d, e \leq k \leq f \}$ is the index set where $a, b, c, d, e, f \in \mathbb{Z}$; the total number of components, $|\{M_{ijk}\}|$, assuming a regular, contiguous network, is $((b-a)+1) \times ((d-c)+1) \times ((f-e)+1)$; and $\forall (i,j,k)$, the component M_{ijk} is specified as

$$M_{ijk} = \left\langle X_{ijk}, Y_{ijk}, Q_{ijk}, \mathrm{I}_{ijk}, \delta_{ijk}, \lambda_{ijk}, T \right\rangle, \text{ where} \tag{B.2}$$

$X_{ijk} = \dot{X}_{ijk} \cup \overline{X}_{ijk}$ is an arbitrary set of input to M_{ijk}, $\dot{X}_{ijk} = \bigcup_{\iota \in I_{ijk}} (\dot{Y}_\iota)$ is the

input into M_{ijk} originating from the set of output of its influencers, $\{\{M_\iota\} \,|\, \forall \iota \in \mathrm{I}_{ijk}\}$, and $\overline{X}_{ijk} \subseteq X_N$ is the input originating from outside the network, N_{CCA}, which is mapped to M_{ijk}. Note that $\overline{X}_{ijk}$ may be $\emptyset$,

$Y_{ijk} = \dot{Y}_{ijk} \cup \overline{Y}_{ijk}$ is an arbitrary set of output from M_{ijk}, where $\dot{Y}_{ijk}$ is the output from M_{ijk} that acts as input to the cells that M_{ijk} influences and $\overline{Y}_{ijk} \subseteq Y_N$ is the output from M_{ijk} that contributes to the network output, Y_N. Note that $\overline{Y}_{ijk}$ may be $\emptyset$,

Q_{ijk} is the set of states of M_{ijk},

$\mathrm{I}_{ijk} \subseteq D$ is the set of indices of the influencers of M_{ijk} where, given a cell M with index (i, j, k), $\mathrm{I}_{ijk} = \bigcup_{a=i-1}^{i+1}(a, j, k)$ for a 1-dimensional network, $\mathrm{I}_{ijk} = \bigcup_{a=i-1}^{i+1} \bigcup_{b=j-1}^{j+1}(a, b, k)$ for a 2-dimensional network, and $\mathrm{I}_{ijk} = \bigcup_{a=i-1}^{i+1} \bigcup_{b=j-1}^{j+1} \bigcup_{c=k-1}^{k+1}(a, b, c)$ for a 3-dimensional network. Note that these are general cases that include M_{ijk} as its own influencer and that $(i, j, k) \notin \mathrm{I}_{ijk}$, implying M_{ijk} is not its own influencer, is permissible.

$\delta_{ijk} : Q_{ijk} \times X_{ijk} \rightarrow Q_{ijk}$ is the state transition function of M_{ijk} that is dependent upon the current state, $q \in Q_{ijk}$ at time t_r, to map the set of component input, X_{ijk} at time t_r, to the new cell state, $q\prime \in Q_{ijk}$ at time t_{r+1},

$\lambda_{ijk} : Q_{ijk} \rightarrow Y_{ijk}$ is the output function of M_{ijk} that maps the cell state, $q \in Q_{ijk}$ at time t_r, to the component output, Y_{ijk} at time t_r, and

T is the network time-ordered set of finite time intervals (defined in equation (B.1)) such that $\forall r, 0 \leq r \leq |T| - 1$, $\{\, \delta_{ijk} \Rightarrow \Delta t \equiv (t_{r+1} - t_r) = h_r \in T \,\}; r, t \in \mathbb{N}_0$;

$h_r \in \mathbb{N}^*$; t_r is the time at the start of the current discrete-time segment; t_{r+1} is the time at the start of the next discrete-time segment; h_r is the time interval between t_r and t_{r+1}; and the time when the network, N_{CCA}, is in its initial state, $q^* \in Q_{ijk},$ is represented by t_0.